Family Mining of State Charts

Master's Thesis

David Wille

November 6, 2014

Institute of Software Engineering and Automotive Informatics
at
Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)

Prof. Dr.-Ing. Ina Schaefer

Abstract

In the automotive or the aviation industry complex software is often created using model-based languages to abstract from the given problem and to easier create a comprehensible solution. Examples for such languages are data flow-oriented languages (e.g., *The Mathworks MATLAB/Simulink* or *ETAS ASCET*) or event-oriented languages (e.g., *The Mathworks Stateflow* or state charts in *IBM Rational Rhapsody*). Since creating these kind of models is still a complex and expensive task, often new variants of an existing product are created by copying the existing functionality and modifying these copies to the changed needs. Maintaining large numbers of related models, which were created using these so-called *clone-and-own* approaches is complicated, as the information about their relationship (i.e., their commonalities and differences) is rarely documented. Family mining solves these problems by automatically comparing a set of selected models and identifying their common and varying parts. During maintenance, this information can be used to understand the relationship between the models, to fix errors in all model variants, or to update some of their functionality to a new version.

The contribution of this thesis is the adaption of the existing family mining approach for block-based languages to state charts. For this adaption, we analyze different state chart notations from different industrial tools and create a common meta-model to store state charts and apply family mining to them. In order to adapt the existing approach, we create additional algorithms to successfully apply it to state charts. Based on the adapted algorithms, we introduce an improved approach for family mining of state charts, which takes advantage of the state chart's characteristics and, thus, performs more efficiently. Both approaches are evaluated and compared using the *BCS SPL case study*, which models the body comfort system for a car using state charts to describe its internal behavior.

Zusammenfassung

Komplexe Software wird in der Automobil- oder Luftfahrtindustrie häufig unter Nutzung von modellbasierten Sprachen entwickelt, um von der gegebenen Problemstellung zu abstrahieren und eine verständliche Lösung zu erstellen. Beispiele für solche Sprachen sind datenflussorientierte Sprachen (wie z.B. *The Mathworks MATLAB/Simulink* oder *ETAS ASCET*) oder ereignisorientierte Sprachen (wie z.B. *The Mathworks Stateflow* oder State Charts in *IBM Rational Rhapsody*). Da die Erstellung solcher Modelle weiter eine komplexe und aufwendige Aufgabe ist, werden neue Varianten eines existierenden Produktes häufig durch Kopieren der existierenden Funktionalität und Anpassung dieser Kopien an die geänderten Anforderungen erstellt. Die Pflege großer Mengen solcher verwandter Modelle, die mittels dieser *clone-and-own* Ansätze erstellt wurden, ist kompliziert, da die Informationen über ihre Verwandtschaft (d.h. ihre Gemeinsamkeiten und Unterschiede) selten dokumentiert ist. Family Mining löst diese Probleme, indem es automatisch eine Menge von ausgewählten Modellen miteinander vergleicht und deren gemeinsame und variierende Teile identifiziert. Während der Wartung dieser Modelle können diese Informationen genutzt werden, um die Verwandtschaft zwischen den Modelle zu verstehen und für alle Modelle gleichzeitig Fehler zu beheben bzw. Funktionen auf einen neuen Stand zu bringen.

Der Beitrag dieser Arbeit ist die Adaption unseres existierenden Family Mining Ansatzes für blockbasierte Sprachen für State Charts. Für die Adaption analysieren wir verschiedene State Chart Notationen industrieller Tools und erstellen für diese ein gemeinsames Meta-Model, das es erlaubt, State Charts zu speichern und den Family Mining Ansatz für sie anzuwenden. Um den existierenden Ansatz zu adaptieren, stellen wir zusätzliche Algorithmen vor, um seine einzelnen Phasen für State Charts anzuwenden. Basierend auf dem existierenden Ansatz stellen wir einen verbesserten Ansatz für Family Mining von State Charts vor, der bestimmte Eigenschaften von State Charts ausnutzt und daher effizienter arbeitet. Wir evaluieren und vergleichen beide Ansätze, indem wir sie auf die *BCS SPL case study* anwenden, die das body comfort system für ein Auto modelliert und dazu State Charts zur Beschreibung des internen Verhaltens nutzt.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Model-based languages are often used in industry to reduce the overall complexity of software in domains with large software systems [2]. Such domains are, for example, the automotive industry or the aviation industry. Model-based languages allow developers to abstract problems to a more comprehensible level and, thus, to reduce the development time and resulting costs. Examples, for such model-based languages are *data flow-oriented languages*, such as *The Mathworks MATLAB/Simulink*[1] and *ETAS ASCET*[2], or *event-oriented languages*, such as *The Mathworks Stateflow*[3] or state charts in *IBM Rational Rhapsody*[4].

A common practice in industry to create variant-rich systems for these model-based languages, is to copy existing products and to modify them to changed needs [2, 17]. These so-called *clone-and-own* approaches lead to different issues and how *family mining* can solve them, which we further explain in Section 1.1. Furthermore, we motivate in Section 1.2, why family mining cannot only be applied to block-based models, but should be adapted to state charts, so that the issues of clone-and-own approaches are also solved for them. In Section 1.3, we outline the research goals, which we follow during this thesis and explain how we approach them. In Section 1.4, we point out the overall contributions of this thesis.

1.1. Motivation

In industry, variant-rich systems are often created, when different customers need the same basic functionality, but have different specialized needs. For example, two customers want to buy an oven. One of the customers needs an circulating air oven to make roast meat, but the other one does not need this functionality and wants to have a grill, in order to make crème brûlée. The basic functionality of the two oven variants is the same (e.g., heating, lighting, control elements, ...), but some of the functionality is different (i.e., in this example the modes of the oven). In industry, such variant-rich systems are often created by applying *clone-and-own* methods [2, 17]. Clone-and-own approaches use copies of existing systems, which were implemented, for example, using block-based modeling techniques, or some high level programming language. The copied files are then modified to changed needs, for example, when a different variant of the system is needed for another customer. These changes can include modifications of elements (e.g., a line of code is changed), deletions (e.g., a line of code is deleted), and additions (e.g., a line of code is added).

In Figure 1.1, we present an example for a system with different variants, which evolved using clone-and-own. The `basic oven` (i.e., with analog control elements, upper and bottom heat, and light) is used as a basis ("cloned") and modified ("owned") to create the two variants `grill oven` (i.e., the `basic oven` with an additional grill) and `circulating air oven` (i.e., the `basic oven` with

[1] http://www.mathworks.de/products/simulink/
[2] http://www.etas.com/ascet/
[3] http://www.mathworks.com/products/stateflow/
[4] http://www.ibm.com/software/awdtools/rhapsody/

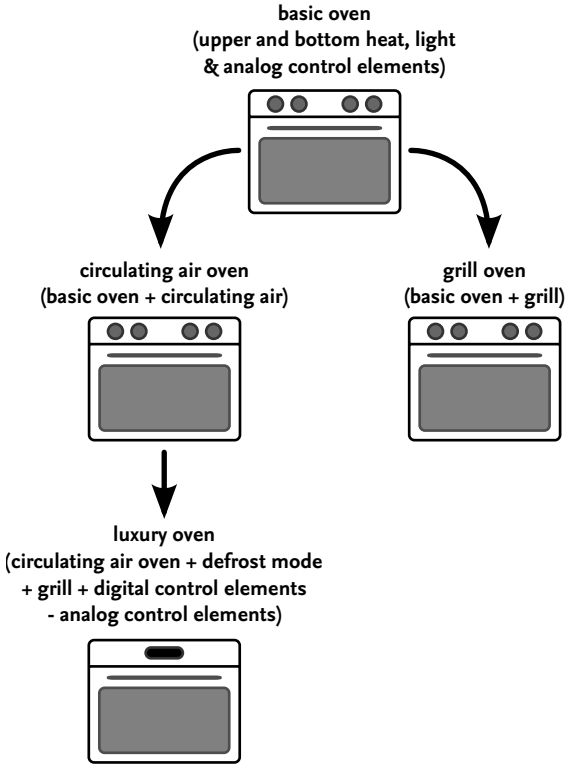


Figure 1.1.: Clone-and-own example

additional circulating air functionality). When creating the `luxury oven`, the developers used the `circulating air oven` variant, added a defrost mode and a grill, and replaced the analog control elements with digital control elements.

The described clone-and-own strategy is very convenient, because it allows to create the desired variants without much effort, because existing functionality can be reused easily. On the other hand, problems arise in the long run, because the information about applied changes between the variants often is not documented. Consequently, developers have problems to understand the relationship between the different product variants and loose the overview over the contained variability and the overall dependencies between these products. They can only be certain, that the copied variant evolved from some specific original variant, but there is no information about the actual changes between these variants. Also, there is not necessarily only one *core variant* for all of the products, because developers use the variant of the existing system, which is the closest to the desired final product (cf., Figure 1.1, where not all variants are created from the `basic oven`).

Consequently, one big issue are errors, identified in one of the variants, because developers cannot tell, which other variants might be compromised, as the different variants have evolved over time and might have changed a lot. Searching for identified errors in the different variants becomes a tedious task, and a lot of reimplementation might be needed to fix the errors in all variants. For example, fixing an error in the `luxury oven` (e.g., the oven always heats up to 20°C above the desired temperature), might be easy, but identifying if the previous variants (`circulating air oven` and `basic oven`) and any variants evolving from them (e.g., the `grill oven`, which evolved from the `basic oven`) contain the same errors, is expensive, since they have to be checked manually. In addition, fixing the identified errors might also be expensive, because the developed bugfix might

not be suitable for all variants, as they evolved over time. The same problems apply, when some functionality should be updated in all variants (e.g., the mechanism setting the temperature has changed, because the corresponding part is used from a different supplier), as the implementation of the functionality might have changed in some variants, when they were adapted to changed needs. Overall, we can see that maintaining different variants of a system, which were created using clone-and-own methods, is a tedious and expensive task.

Another issue with clone-and-own, is the reuse of parts in new variants. We cannot reuse all implementation artifacts, since we can only use one variant as a basis for new variants. For example, when creating the `luxury oven`, we can only use the `circulating air oven` variant *or* the `grill oven` variant as a basis, but not both at the same time, because they cannot be merged easily. Consequently, we have to decide, which variant we want to use as a basis and reimplement the functionality, which we cannot adapt. In our example, we would have to reimplement the grill functionality. Of course, this is not desired, because this approach is inefficient, and it would be more convenient, if we could reuse every functionality, which is already created for another variant.

To overcome all these issues, *family mining* is used to automatically identify the commonalities and differences between different variants of a system, which evolved using clone-and-own strategies [6, 24]. By automatically comparing the variants and their implementation artifacts (e.g., the inner components of the oven) with each other, the variability between them is identified. After comparing the variants and identifying the variability, a fine-grained overview of the results can be created, which helps developers to understand the changes and dependencies between variants and eases maintenance of the created software. With suitable generators, such a representation could also be used to generate all compared variants from the identified artifacts, or even new variants combining the artifacts from different variants. Consequently, the reuse of developed artifacts is improved, because they can be reused more easily.

1.2. Family Mining of State Charts

Both language types, data flow-oriented languages and event-oriented languages, allow to model complex systems with different means. Data-oriented languages normally use atomic blocks and connectors to create systems, which process data introduced to them via inport blocks. Thus, these models are also called block-based models. The data passed to the models is processed by the functions of the atomic blocks, which usually use mathematical operators to alter data, that is passed to them via incoming connectors. After processing the data, the corresponding results are emitted via their outgoing connectors and after traversing the complete system, the final results are emitted via its outports. Using the connectors between the blocks the data flow is modeled and complex systems can be created. These kind of models are, for example, used to represent complex control circuits for driver assistance systems in cars.

An example for such a system is an *adaptive cruise control (ACC)* system, which automatically keeps a specified distance to other cars and does not exceed a defined maximum speed. This system is realized as a control circuit to continuously adjust the car's speed and its distance to other cars by processing the inputs from sensors (e.g., distance sensors and speed sensors) to influence the system's behavior by controlling its actuators (e.g., the breaks and the accelerator). The sensor inputs are processed by the blocks of the model, which calculate the corresponding control variable to change the behavior of the actuators and correspondingly the behavior of the car.

Event-oriented languages, on the other hand, use states and transitions to model their systems with state charts. A state represents an execution state in the system and allows to alter the system's variables by different actions. Starting at an initial state, the data is processed in the different states of the system. Events from outside of the system trigger state changes via the system's transitions, which can also modify the system's variables by their actions.

An example for such an event-driven system could be a windscreen wiper in a car, which is controlled with a setting lever. In the initial position of the lever, the state chart is in the initial state and the wiper is stopped. When the user changes the position of the lever an event is triggered and the movement of the wiper is started. Depending on the mode selected by the current lever position, the speed of the wiper is changed or stopped. All windscreen wiper modes are represented by states and changing the lever position triggers events, which provoke the corresponding state changes via the state chart's transitions.

For both model types, data flow-oriented models and event-oriented models, clone-and-own approaches are applicable to create related variants of the same systems. For example, the created control circuit for the ACC system might need to be modified to be usable in another car. Also the software for the windscreen wiper might need to be change, when, for example, a different windscreen is used or it is build in another car. For both examples, creating a copy of the existing software and modifying it to the changed needs (i.e., applying clone-and-own approaches) is a quick and cheap solution, since the common parts can be reused and only small parts have to be modified. Thus, this approach is commonly used in industry for both model types.

These clone-and-own approaches bring drawbacks along, which can be solved by family mining. We argue, that data flow-oriented models and event-oriented models are closely related according their basic structure, because both use nodes and edges to represent their functionality. Similar to block-based models, different variants of state charts can be created using clone-and-own approaches. Consequently, we argue that adapting our current family mining approach for block-based models [7] to state charts is sensible and should help to solve the discussed clone-and-own issues for these models.

1.3. Goals and Approach

As our main goal is to adapt our current family mining approach for block-based models to state charts, we need to find solutions to address the following *research goals (RGs)*. Below each of these *RGs* we present the basic approach, which we follow to address them.

Research Goal 1: *Find a generic state chart representation for different state chart notations from literature and industry and compare their notations.*

For *RG1* we have to create a common meta-model for state chart notations from different industrial tools and notations used in the literature. This meta-model is needed to transfer state charts from these varying notations into a generic format, which is the basis for the adaption of the existing family mining approach for block-based models to state charts. Thus, instead of adapting the existing approach for one of the notations, we are able to create a common approach, which allows to apply the adapted family mining algorithms to all notations summarized in this meta-model. In order to create the common meta-model for state charts, we analyze and compare the notations from

Harel [5], who first defined state charts, with the notations from *The MathWorks Stateflow R2012b*[5], *ETAS ASCET 6.1.3*[6], *IBM Rational Rhapsody 8.0.6*[7], *Esterel Technologies SCADE Suite R15a*[8], and the *UML specification* [15].

Research Goal 2: *Identify the differences and commonalities between block-based models and state charts.*

RQ2 is a prerequisite for *RQ3*, since identifying the differences and commonalities between these notations helps to adapt the existing algorithms. First of all, we identify whether elements from the two notations are similar to each other. For these elements, we have to check if the corresponding compare algorithms can be adapted from block-based models to state charts or if new ones have to be created. Furthermore, we identify elements, which are not contained in block-based models and, thus, require to create new algorithms in order to compare them with each other.

Research Goal 3: *Adapt the existing family mining approach for block-based models to state charts.*

For *RG3*, we adapt the existing family mining approach for block-based models [7] to state charts and use the insights, which we obtained from *RG2*. We adapt all phases and their algorithms, which are used during the family mining of block-based models. We have to adapt the compare algorithms for model elements, which were identified in *RG2* to be similar for block-based models and state charts. Furthermore, we have to create new algorithms for all elements, which are not contained in the existing approach.

Research Goal 4: *Investigate whether we can improve the adapted algorithms for state charts, in order to process them more efficiently.*

After adapting the existing approach for block-based models to state charts, we investigate whether the adapted approach can be improved by taking advantage of certain state chart characteristics, in order to execute the family mining more efficiently. Examples could be elements, which have a different meaning or importance in state charts compared to block-based models and, thus, change the way we have to execute the family mining algorithms.

Research Goal 5: *Evaluate whether the adapted family mining approach for state charts and possibly improved variants of this approach create correct results.*

After adapting the existing family mining approach to state charts and creating possibly improved variants of these algorithms, we check the correctness of the created results by evaluating them with state charts from a case study.

Research Goal 6: *Evaluate the performance of the adapted family mining approach for state charts and possibly improved variants.*

During the analysis of the correctness, we gather measures, which allow to evaluate the performance of the adapted family mining approach. For improved variants created in *RG4*, we can also gather these measures and utilize them to compare the performance of these differing family mining approaches.

[5] http://www.mathworks.com/products/stateflow/
[6] http://www.etas.com/ascet/
[7] http://www.ibm.com/software/awdtools/rhapsody/
[8] http://www.esterel-technologies.com/products/scade-suite/

1.4. Contribution

In this thesis, we adapt the existing family mining approach for block-based models [7] to state charts. First, we analyze and compare different state chart notations to create a common meta-model for them. The created meta-model is used to store state charts, which should be compared using the adapted family mining approach. During the adaption of the existing family mining approach for block-based models to state charts, we not only adapt the existing algorithms, but extend it with further algorithms in order to be able to compare all contained elements. Furthermore, we improve the adapted algorithms by using characteristics common to state charts, which allow to execute the family mining algorithms more efficiently. The results created by both family mining approaches are evaluated regarding correctness and performance by using the state charts from the *BCS SPL case study*.

Consequently, the contribution of this thesis is as follows:

- We compare different state charts notations from literature and industry.
- We create a common meta-model representation for different state chart notations.
- We adapt the existing family mining approach for block-based models [7] to state charts.
- We improve the adapted family mining approach to be more efficient by taking advantage of certain state chart characteristics.

1.5. Outline

This thesis is structured as follows: In Chapter 2, we explain the details of block-based models and the output format of our current family mining approach. These details are needed to explain our previous work [6, 24] and our current family mining approach [7] for block-based models. Next, we analyze different state chart notations in Chapter 3 to find a way, how we can abstract from these notations to a common meta-model. This meta-model is used to store models, which we want to compare with our family mining algorithms. Correspondingly, we also analyze the differences between block-based models and state charts, in order to better understand the commonalities and differences between them to adapt the current family mining approach to state charts in the next section. In Chapter 4, we explain the changes, which have to be applied to the current algorithms, in order to successfully apply family mining to state charts. Besides, we introduce a new algorithm for family mining on state charts, which takes advantage of certain state chart characteristics to compare them more efficiently. In Chapter 5, we explain the details of our current implementation for family mining of state charts and how we realized the adapted and the new algorithm. This implementation and the underlying algorithms are evaluated in Chapter 6 by applying them to state charts from a case study. We evaluate, whether the results created by the algorithms are correct and how the adapted algorithm and the new algorithm perform compared to each other. Finally, in Chapter 7, we give a summary of our results and present future work to further improve the family mining approach for state charts.

2 Background

In this chapter, we give an introduction to the notion of *block-based models* (cf., Section 2.1). In Section 2.2, we explain *feature models* and *family models*, and why we chose family models over feature models to store the results of family mining. In the following section, we explain two previous approaches, which we used for family mining of block-based models (cf., Section 2.3), and which lead to our current approach, which we explain in Section 2.4.

2.1. Block-based Models

In this section, we describe block-based models, which are expected by our previously realized approaches [6, 24] and our current approach. The basic elements described in this section are used by all block-oriented modeling languages, such as *The MathWorks MATLAB/Simulink*[1], *ETAS ASCET*[2], or *Esterel Technologies SCADE Suite*[3].

Block-based models consist of atomic blocks, which have a name and some *functionality*. This functionality is commonly defined by code in some high level programming language (e.g., C code), or a block type provided by the library of the used block-oriented modeling language (e.g., a `Gain` or `Sum` block in *The MathWorks MATLAB/Simulink*). According to the defined functionality these blocks process data, which can be submitted to them through their *inports*. The results of the calculation are emitted via their *outports*. These ports define the blocks' *interfaces* [24]. In order to exchange data between the blocks, connectors exist and can be used to connect the blocks' ports with each other. Besides, the mentioned attributes, defining the blocks' identity (i.e., name, functionality, and interface), other parameters exist, that can be neglected during the family mining (e.g., the blocks' color, or their location in the model), because these attributes do not affect how these blocks behave. In order to allow different levels of abstraction, most block-oriented modeling languages allow to create arbitrarily nested hierarchies by adding blocks to other blocks.

In Figure 2.1, we present an example for a small block-based model. This model, represents the addition of two values. As we can see, the two values, which should be added together, are emitted to the model via the blocks `Input1` and `Input2`. The emitted values are transfered to the block `Sum` via the outgoing transitions of both blocks and the `Sum` block processes the values (i.e., adds them together) and emits the final result via its outgoing transition to block `Output1`.

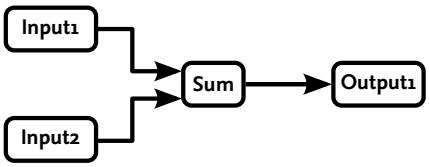


Figure 2.1.: Example for a block-based model

[1] http://www.mathworks.com/products/simulink/
[2] http://www.etas.com/ascet/
[3] http://www.esterel-technologies.com/products/scade-suite/

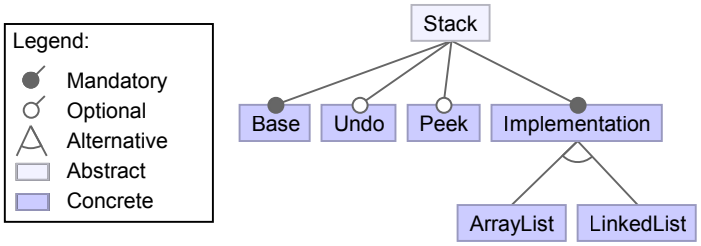


Figure 2.2.: Example for a feature model

2.2. Feature and Family Models

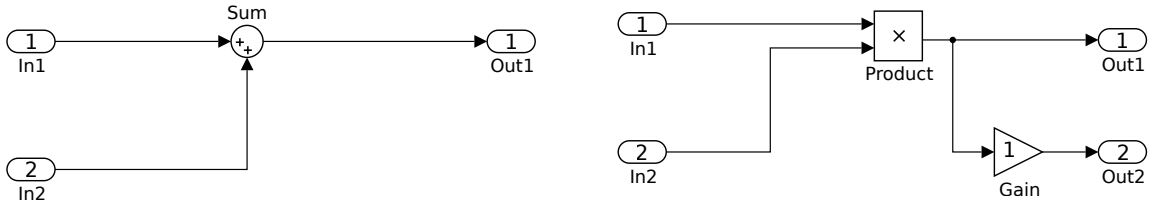
In this section, we introduce and compare *feature models* and *family models*, which are different notations for variant-rich systems. These visualizations are commonly used to model systems with different configuration options. Furthermore, we argue, why we use family models over feature models to represent the results of the family mining.

2.2.1. Feature Models

Czarnecki et al. [1] introduce *problem space* and *solution space* as notions for variant-rich systems. The problem space models which *features* can be added to a system, but does not give any details about their concrete implementation. Features are labels, which define possible configuration options for a system with different variants [1]. A good means to model the variability in the problem space, are *feature models*, because they capture the dependencies between these features. In Figure 2.2, we present the feature model for a small `Stack` product line, which allows to configure different stack systems by selecting and deselecting features. As we can see the `Stack` has two *mandatory* features (i.e., features that have to be included in every legal configuration of the system), `Base` and `Implementation`. In addition, the feature model defines, that we can select the features `Undo` and `Peek`, which allow to undo the last step (`Undo`) and to get the first element on the stack without removing it (`Peek`). These *optional* features are part of the system's variability, because they do not necessarily have to be part of the possible configurations. As the `Implementation` feature is mandatory, we have to select either an implementation using an `ArrayList` or a `LinkedList`. These features are *alternative*, because the relation between them defines mutual exclusion (i.e., at least on of these feature has to be selected, but not more than one). Thus, they are part of the systems variability, because they allow to configure different variants. In addition to the mentioned possibilities to configure the system, there are also *or* relations, which define that at least one feature has to be selected. As we can see, none of the features in Figure 2.2 shows concrete implementation details, because they are only labels, which define the *configuration space*. Consequently, feature models only show the variability of different system variants abstractly and how the features can be combined to create these variants.

2.2.2. Family Models

In contrast to feature models (cf., Subsection 2.2.1), family models define the solution space and give a complete overview about the implementation-specific variability. The solution space shows the details of the implementation and which concrete implementation artifacts define possible variants. These family models are a view for *annotated 150% models* (or for short *150% models*), which



(a) `Model 1` (b) `Model 2`

Figure 2.3.: Two related models, created with the clone-and-own approach

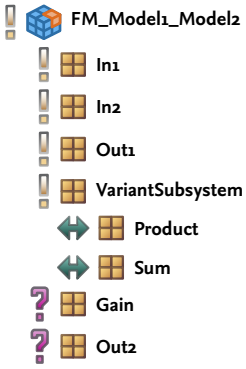


Figure 2.4.: Example for a family model

contain not only the implementation artifacts for one product variant, but for all possible system variants. The artifacts in the 150% model are annotated with information about their variability and traceability across the elements is enabled. Thus, developers can easily understand which implementation artifacts belong to the different system variants. We argue, that family models are well-suited to overcome the problems introduced by clone-and-own approaches, because the work of developers is improved, since they get an overview about the system's variability with all concrete implementation artifacts. For example, these models allow them to fix errors, that are affecting different variants of the overall system, by fixing the errors in the corresponding implementation artifacts.

The two models in Figure 2.3 were created using the clone-and-own approach. As we can see, the `Sum` block in `Model 1` is replaced by the `Product` block in `Model 2`. In addition to that, another outport `Out2` and a `Gain` block were added to `Model 2`. An algorithm comparing these two models, should return the family model in Figure 2.4. As we can see, the alternative blocks `Sum` and `Product` are identified, and marked accordingly with "⇔". They are added to the `VariantSubsystem`, in order to group them together. The two optional elements `Gain` and `Out2` from `Model 2` are marked with "?", to indicate that they do not have to be part of every system variant. All other mandatory elements (i.e., `In1`, `In2`, and `Out1`) are marked with "!" to indicate, that they have to be part of all system variants. The representation used in Figure 2.4 is based on the representation used by *pure::variants*, which is a framework for developing and managing software product lines (SPLs) distributed by *pure-systems*[4]. This is also the representation for family models, which we will use throughout this thesis.

[4] http://www.pure-systems.com/

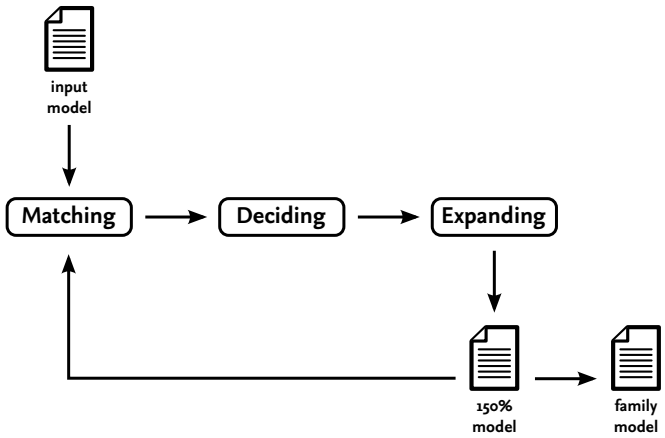


Figure 2.5.: Workflow for the approach in [6]

2.3. Previous Family Mining Approaches

In this section, we will describe previous approaches [6, 24], that we used to apply family mining to block-based models. In Subsection 2.3.1, we explain our first approach, which had drawbacks regarding the number of compared blocks and the supported differences in the models. In Subsection 2.3.2, we show the improvements of our second approach and explain its drawbacks and why we had to improve it, resulting in our current approach, explained in Section 2.4.

2.3.1. Automatic Synthesis of Family Models by analyzing block-based Function Models

The approach in [6] consists of a workflow with three phases *Matching Phase*, *Deciding Phase*, and *Expanding Phase*. In Figure 2.5, we present this workflow. In the *Matching Phase*, the blocks from two model variants are compared and matched according to there structural similarity. The approach creates pairs of blocks by comparing every block from the first model variant with all blocks from the second model variant, which were not matched previously. For the approach in [6], we defined that the first model variant is always the model, which was selected first. During the comparison two structural attributes are considered. First, the interface (i.e., the number of in- and outports) is compared and the two blocks are considered as not similar, when their interfaces differ (i.e., one of the blocks has a different number of in- or outports than the other). Consequently, this approach lacks support for blocks with differing interfaces, but similar functionality (cf., the approaches described in Subsection 2.3.2 and Section 2.4, which overcome this limitation). The second attribute is the block's neighborhood, which describes the context of the blocks and the blocks connected with this context. Blocks, which are closer to the compared blocks get a higher weight in the neighborhood similarity than blocks with a larger distance. For the neighborhood similarity only the interface of the neighbors and their block types (i.e., functionality) are considered. The pair of compared blocks with the highest similarity is removed from the list of non-matched blocks and added to the list of matched blocks. Consequently, the approach terminates, since the list of non-matched blocks has to be empty at some point.

In the *Deciding Phase*, all matched pairs are processed and their variability is added to the first model variant. When the pair's blocks are the same, they can be considered as mandatory elements,

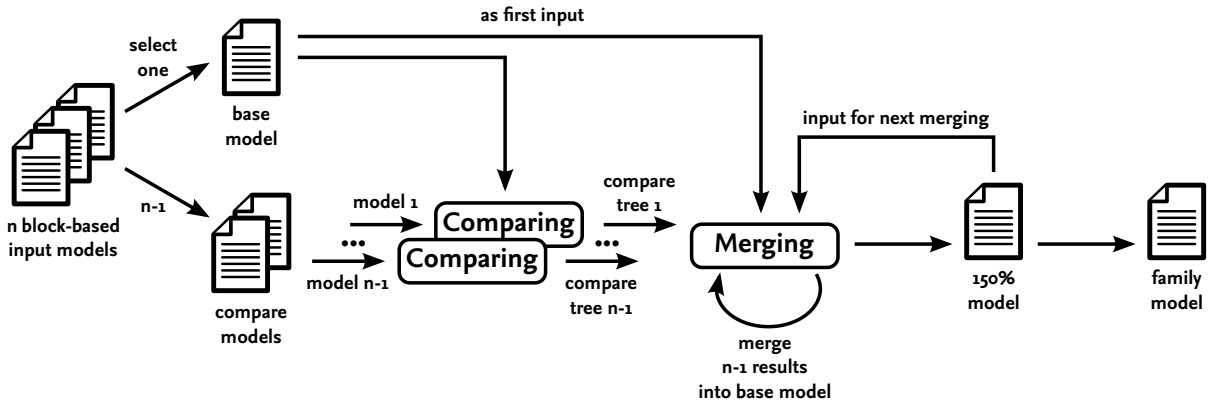


Figure 2.6.: Workflow for the approach in [24]

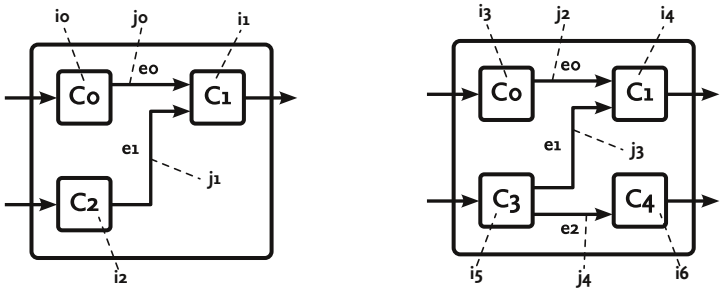
since they exist in both model variants. Consequently, no additional steps are needed for these blocks and they can be marked directly as mandatory blocks. In contrast, all blocks, which are not the same, are considered as alternatives. Therefore, the pair's block, which does not exist in the first model variant, has to be added, in order to represent the correct variability. These alternatives are represented by creating a subsystem, which contains the block from the first model variant and the newly added block as variants. Both blocks are marked as alternative blocks. As the compared models can have different sizes, there can be left over blocks, which were not matched to another block in the other model. These blocks are considered and marked as optional blocks and can be added directly to the first model variant, if they do not already exist (i.e., if the block is an optional block from the model compared with the first model variant).

Finally, during the *Expanding phase* subsystems containing alternative blocks are expanded to bigger subsystems. Starting from a subsystem containing alternative blocks all neighboring blocks are considered. If one of the neighbors is also a subsystem containing alternative blocks, the subsystem will be integrated into the considered subsystem. This step is executed for all subsequent blocks, until a mandatory block is found, or another block, which was already processed. After this last step, a 150% model exists, which can be used as an input for the next model comparison, or exported to a family model representation.

2.3.2. Interface Variability in Family Model Mining

To overcome the limitation of our first approach in [6], that no blocks with differing interfaces can be compared, we created the approach in [24]. This approach also changes the way, how we process the blocks from the different models. In Figure 2.6, we present the work flow, for this approach.

In order to explain our approach, we will compare the models in Figure 2.7. When starting the comparison, we label their blocks and connectors with unique identifiers (i.e., for our example, `i0`, `i1`, ..., `i6` and `j0`, `j1`, ..., `j4`, respectively). The next step is to select a base model from the list of models, that should be compared. For this approach, it is defined, that the *base model* is always the model with the highest number of blocks (i.e., in our example, the model in Figure 2.7b), because it enables the approach to identify optional elements. All other models are *compare models*, which are compared with this model. For every compare model, the comparison is started by comparing each of its *start blocks* with a start block selected from the base model at the start of the comparison (i.e., in our example `i3`). These start blocks are the input blocks of a model and are the initial entry point



(a) `Model 1` (b) `Model 2` (base model)

Figure 2.7.: Example models used to explain the approach

for data, that should be processed by the model (i.e., `i0`, `i2` and `i3`, `i5`). When comparing two blocks with each other a so-called *compare element* is created, containing the two compared blocks and their similarity value (i.e., a value between 0 and 1 representing the blocks' similarity), calculated according to a metric (e.g., comparing `i3` with `i0` results in a similarity of 100%). A compare element, is represented by a dashed circle (i.e., the element from the base model), a solid circle (i.e., the element from the compare model), and a line connecting these two circles, showing their similarity and that they were compared. The metric defines, which attributes of blocks should be considered and what weight these attributes get in the overall similarity (e.g., the block's functionality has a higher weight than the names). Besides, the metric defines, that interfaces are compared by calculating the average between the neighbors, which are the same and the overall number of neighbors. Consequently, blocks with differing interfaces can be compared by this approach.

All created compare elements are stored in a so-called *compare tree*, which documents all comparisons and shows the order of the executed comparisons. These compare trees can have different branches, when there are different matching possibilities. In Figure 2.8, we show the compare tree for the comparison of the two models in Figure 2.7. In our example, there are two start blocks in the compare model (i.e., `i0` and `i2`), and thus, we have to create two branches with all subsequent comparisons, since both elements can represent a good match for block `i3`. Beside these branches, we have to create branches, when more than one subsequent block is connected to a compared block.

After comparing the start blocks of the models, we create a list of all successors and predecessors of the compared blocks by analyzing the data flow and following the blocks' transitions. This list contains only possible candidates for the next comparisons, and we have to remove all elements, which were previously compared. For example, in a later phase of the comparison, the analysis of possible candidates in component `i4` returns the components `i3` and `i5`. As component `i3` is already used in a previous comparison, we do not take it into account during following comparisons.

The resulting list is used for the next comparisons by comparing the base model blocks with the compare model blocks. This step is repeated, until no new compare elements are created and every block has been considered in every branch. During the comparisons it is possible, that no new blocks are found in the compare model, which are compared with the block from the base model. Consequently, these base model blocks are compared with `null`, as they represent optional elements (e.g., the comparison of `i6`). After finishing the comparisons, the similarity of the different branches in the compare tree is calculated and the branch with the highest similarity is selected as the result of the comparison.

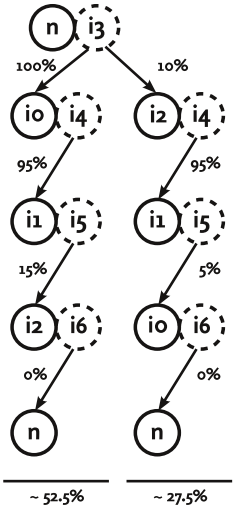


Figure 2.8.: Compare tree comparing the blocks of the two models

With the selected branch, it is now possible to create the 150% model by processing the elements and storing the identified variability according to their similarity in the base model. The variability is interpreted by using thresholds, which are also defined by the metric and identify, whether two compared elements are mandatory, alternative, or optional. The created 150% model is used as a new input for the merging of all subsequent compare trees. After merging all results into one 150% model, this model can be exported into a family model.

The presented approach from [24] still has two drawbacks, which are considered in the previous approach (cf., Section 2.4). First, the data structure used to store the intermediate results is traversed repeatedly during the comparisons (e.g., when checking whether a block is already considered in all branches), which would not be efficient for industrial-scale models. Second, the approach is not able to find all optional elements and only identifies them, when they are located at the end of the data flow. However, the approach gives a good idea, how blocks with differing interfaces can be compared using the right metric.

2.4. Current Family Mining Approach

In this section, we introduce the improved family mining approach [7]. This approach expects block-based models as described in Section 2.1 and consists of a workflow with three phases [7]. In Figure 2.9, we present this workflow. Before starting the *Comparing Phase* an arbitrary *base model* has to be selected from the input models. In the *Comparing Phase* (cf., Subsection 2.4.1), the blocks in the remaining *compare models* are compared with the blocks in the base model and *compare elements* representing possible matches are created. These elements compare two blocks and store their calculated similarity. Using the created list of possible matches during the *Matching Phase* (cf., Subsection 2.4.2), we find distinct matches for the blocks of each compare model according to the blocks from the base model. With the list of distinct matches, we are able to determine the variability in the *Merging Phase* (cf., Subsection 2.4.3). In this phase, we use a copy of the base model and merge it with the matched results of the first compare model and annotate the variability. The resulting annotated 150% model, containing all common and varying parts of the compared models, is used

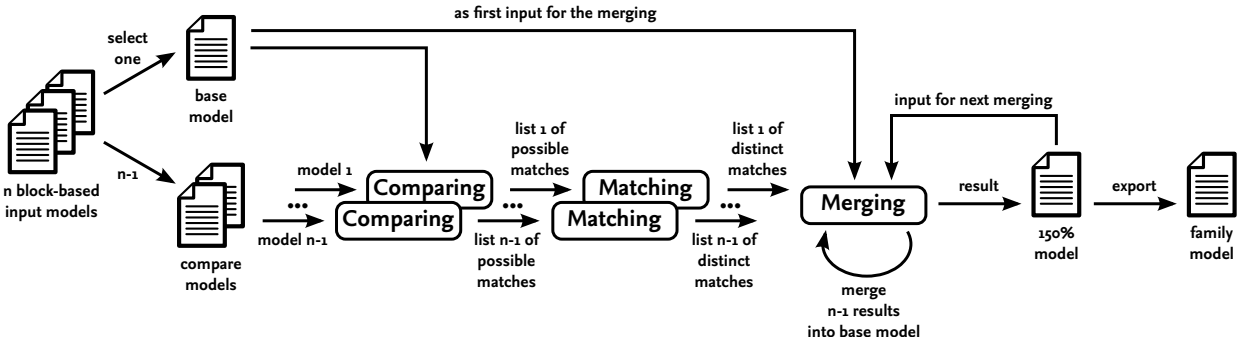


Figure 2.9.: Workflow for the current family mining approach in [7]

as an input for the merging of the next compared model. After merging the determined variability of all models, we can export the resulting 150% model to some family model representation (e.g., with pure::variants). In order to illustrate the approach, we will compare the two *The MathWorks MATLAB/Simulink* models in Figure 2.3 in the following three subsections.

2.4.1. Comparing Phase

Prior executing the comparison of multiple models an arbitrary *base model* has to be determined. The other models are so-called *compare models* and are compared with the previously selected base model (e.g., `Model 2` for our example). The comparison of two models is started by selecting the *start blocks* from the base model and comparing them with all start blocks from the compare model. These start blocks are input blocks of a model and provide the initial access to the data flow and thus are a perfect starting point for our data flow-oriented comparison of related models. For both models in our example the blocks `In1` and `In2` are the only start blocks. When comparing a block from the base model with a block from a compare model a so-called *compare element* is created. Such a compare element stores the blocks from the base model and the compare model with a *similarity value* according to a *metric*. This metric holds different weights for the properties of the blocks and allows to calculate the similarity value of the compared blocks in the interval from 0 to 1. The metric can be adapted to the corresponding model type (e.g., *The MathWorks MATLAB/Simulink* or *Esterel Technologies SCADE Suite*) and domain-specific knowledge according to different use cases. Properties considered in such a metric are the blocks' names, functions and their interfaces (i.e., the number of similar in- and outports). During the comparison of the blocks' interfaces, we compare the blocks' neighbors (i.e., predecessors and successors connected to the compared blocks via connectors). When comparing these neighbors only the names and functions of the blocks are considered and we calculate the fraction of similar neighbors to the overall number of neighbors. The overall similarity stored in compare elements shows how similar the compared blocks are.

After comparing the start blocks of the compare models with the start blocks from the base model, we follow the data flow and create the lists of subsequent blocks for these blocks (i.e., `Sum` for `Model 1` and `Product` for `Model 2`). Each of the subsequent blocks from the base model is compared with each block from the list of subsequent blocks in the compare models, creating new compare elements. This step is repeated until no new blocks are found when following the data flow and, thus, the comparison does not create new compare elements. When comparing models with different sizes (i.e., different number of blocks), it is possible, that we do not find any new subsequent blocks

in the compared models. Consequently, each of the remaining model blocks is compared with `null`, as these blocks might represent optional elements. Such a comparison with `null` implies that the corresponding block might represent an additional element, which will be marked as optional in the resulting family model. For the two related models in Figure 2.3 the resulting list of compare elements is: `(In1,In1)`, `(In1,In2)`, `(In2,In1)`, `(In2,In2)`, `(Sum,Product)`, `(Gain,Out1)`, `(Out1,Out1)`, and `(Out2,null)`

2.4.2. Matching Phase

In the *Matching Phase*, for each compared model, we walk through the list of possible matches, created during the *Comparing Phase*. Iterating over these lists, we try to find distinct matches for each block compared with a block from the base model. While walking through the list of compare elements, we check whether other compare elements exist that contain either the same base model block or the same compare model block. If no other compare element exists, we can directly match the two blocks from the corresponding compare element (e.g., `(Product,Gain)`). If we find other compare elements containing the same blocks, we select the compare element with the highest similarity and delete all ruled out compare elements from the list of possible matches. For example, when looking at `(Gain,Out1)`, we also find `(Out1,Out1)`, which has a higher similarity, since the blocks' type and name are the same. Consequently, we match `(Out1,Out1)` and delete the ruled out compare element `(Gain,Out1)` from the list of possible matches.

Sometimes, we have situations, where we cannot directly match blocks, because there exist other relevant compare elements with the same similarity value. Consequently, no distinct match is possible and we move these ambiguous elements to the end of the list and continue matching, hoping that other matches resolve these conflicts. If the conflict cannot be resolved and we revisit these ambiguous elements, we have to present these problematic compare elements to the developers, in order to get a manual decision, which is the most appropriate match.

After processing all compare elements, there are certain cases, where we did not match every block with another block from the other model or `null`. For example, during the matching of `(Out1,Out1)` the compare element `(Gain,Out1)` is deleted from the list of possible matchings, since it is ruled out because of the lower similarity value. For each of these blocks without a matching partner, a new compare element comparing the corresponding block with `null` is created. The resulting final list of matched blocks for our example is: `(In1,In1)`, `(In2,In2)`, `(Sum,Product)`, `(Out1,Out1)`, `(Out2,null)`, and `(Gain,null)`

2.4.3. Merging Phase

In the final *Merging Phase*, the 150% model is created, which afterwards can easily be exported to a family model representation (e.g., using pure::variants). In order to create this 150% model, we use a copy of the base model and process the list of matched blocks for the first model from the previous phase. During the processing, we interpret the similarity value of all compare elements, which was calculated during the *Comparing Phase* according to the metric used to compare the blocks with each other. In order to distinguish between common elements (i.e., mandatory elements) and differing elements (i.e., alternative and optional elements), we use different thresholds defining how to interpret the similarity values of the compare elements. For example, such a threshold could define that a compare element with a similarity value of 95% or more contains mandatory blocks

and that a compare element with a similarity value of 0% is optional. All other compare elements with 0% < x < 95% would be considered as alternative blocks. Similar to the metric, these thresholds can also be adapted to different model types and use cases.

Using the determined variability information, we can mark the blocks in the 150% model accordingly. For mandatory and optional elements, we can mark the blocks directly, whereas for alternative elements we have do some extra work. First, we have to create a so-called *VariantSubsystem*, which is marked mandatory and later contains the possible variability. Second, we copy the elements, that are alternatives to each other, from their current hierarchy level into this VariantSubsystem and mark them as alternatives. The resulting 150% model is used to merge all other models, one after another into the final 150% model. For the two models in Figure 2.3, we detect that `In1`, `In2`, and `Out1` are mandatory elements, since they are contained in both model variants. `Product` and `Sum` are detected as alternative elements and are moved to a mandatory VariantSubsystem accordingly. The two blocks `Gain` and `Out2` are marked as optional elements, since they are only contained in `Model 2`.

After creating the final 150% model, we can easily export the mined information about commonalities and differences between the compared models and create the corresponding family model. The resulting family model for our example is shown in Figure 2.4.

2.5. Summary

In this chapter, we presented block-based models, which are used as inputs for our previous family mining approaches (cf., Section 2.3) and our current family mining approach (cf., Section 2.4). We created the current approach, because our previous approaches had certain limitations. The first approach cannot compare models, which contain blocks with different interfaces. The second approach has limitations regarding the performance and its results, because it processes the used data structure repeatedly during the comparisons and certain variability is not identified correctly (i.e., some optional blocks are not identified). In addition, to the presented block-based models as inputs, we also compared feature models and family models and argue, that family models should be chosen as output format over feature models. Main reason is, that they give an overview about the variability of the implementation artifacts and not only the configuration options of the system.

3 Analysis of State Chart Notations

In this section, we describe the basic elements of state charts, as they were defined by Harel [5], and extend these with elements utilized by four tools commonly used in the industry to create state charts. These tools are *The MathWorks Stateflow R2012b*[1], *ETAS ASCET 6.1.3*[2], *IBM Rational Rhapsody 8.0.6*[3], and *Esterel Technologies SCADE Suite R15a*[4]. We choose these tools for our analysis, because all of them allow to create state chart representations and are commonly used in industry (e.g., the automotive, aviation, and transportation domains) to model large-scale systems. Consequently, by choosing these tools, we consider highly used and relevant tools, from domains, where large systems are developed and clone-and-own strategies are often applied. In addition to these tools, we consider the elements introduced by the *UML specification* [15]. By extending Harel's [5] notion of state charts with elements commonly used in industry tools and the *UML specification* [15], we create an overview of utilized elements, that is as complete as possible, so we can use this information to create a suitable meta-model representation for a wide range of state chart types. Consequently, we will only list all possible elements introduced by Harel [5], the *UML specification* [15], and the four tools, rather than comparing the notations and pointing out, which tools provide certain possibilities or limitations.

State charts are commonly used to express an abstract description for the behavior of a system in a state-based manner [5]. A *state* represents an execution state of the corresponding system. The current state can change when a *transition* to another state exists and the *event* and the *condition* defined by the transition are met. In contrast to the definition by Harel [5] and *IBM Rational Rhapsody*, all other considered tools and the *UML specification* [15] use the term *state machine*. In the course of this thesis, we will use the term state chart.

State charts are a notion to model *deterministic finite automatons (DFAs)* and extend their semantics with further elements [5]. DFAs model systems with a finite set of *states*, which need to include exactly one *start state*, which shows, where the execution of the system is started [8]. In addition, DFAs can also define *accepting states*, which end the execution of the system [8]. A DFA can only be in one state at the same time and consequently, does not allow parallel execution. Between states of a system *transitions* can exist, which allow to move from one state to another, in response to some external *input* [8].

In Figure 3.1, we present a small example for a DFA. It represents, a system, which describes the modes of a *network attached storage (NAS)* system and consists of three states `Standby`, `Off`, and `On`.

[1] http://www.mathworks.com/products/stateflow/
[2] http://www.etas.com/ascet/
[3] http://www.ibm.com/software/awdtools/rhapsody/
[4] http://www.esterel-technologies.com/products/scade-suite/

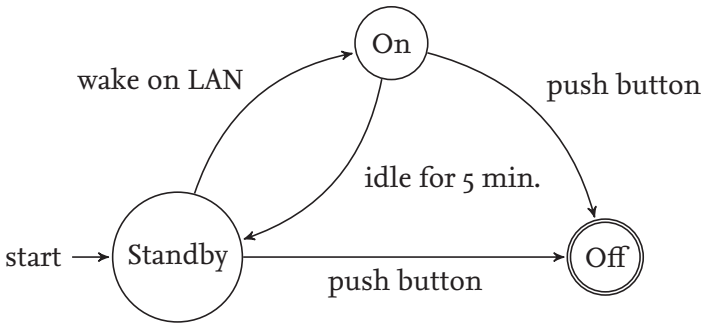


Figure 3.1.: Example for a deterministic finite automaton (DFA)

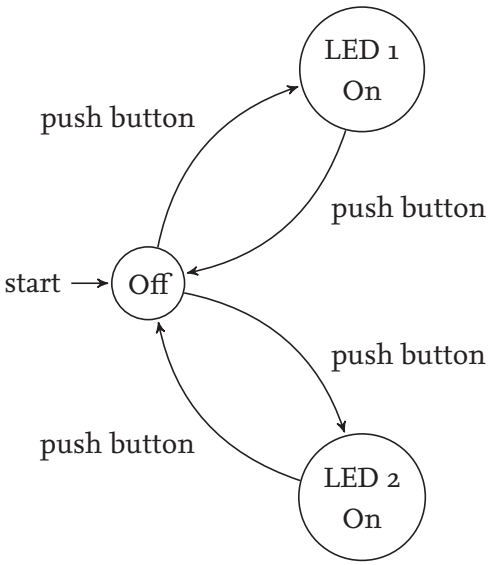


Figure 3.2.: Example for a nondeterministic finite automaton (NFA)

At the execution start (i.e., when the system is plugged into a power outlet) the NAS system powers on and enters the `Standby` state. Every time, the NAS system receives a "wake on LAN" command, it enters the `On` state and wakes up from the standby mode. When the system is idle for 5 minutes, the system enters the `Standby` state again. In both modes, it is possible to push a button on the NAS system and power the system off (i.e., the accepting `Off` state is entered).

At every state of a DFA, there cannot be more than one outgoing transition with the same input event. Otherwise, the execution of the system would be *nondeterministic*, because it would not be clear, which transition has to be taken, when the corresponding input event occurs [8]. Such a system would be a *nondeterministic finite automaton (NFA)*. In Figure 3.2, we present an example, for such an NFA, which allows to turn two LEDs on and off. When pushing the button in the start state `Off`, the NFA nondeterministically decides, which transition it takes, and whether it turns on `LED1` or `LED2`. Consequently, we cannot tell, which of the LEDs is turned on, when we push the button. When pushing the button again, the nondeterministically chosen LED is turned off.

Harel [5] describes how state charts can model DFA and extends their semantics with further elements, such as parallel states to model parallel systems, hierarchical states to reduce the complexity of the systems, and state activities to execute actions when entering a state [5]. All these elements

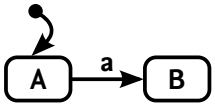


Figure 3.3.: State charts

are discussed in Chapter 3 and further extended with other elements introduced by a number of industrial tools, which allow to create such state charts.

We explain states and special notations for certain state types in Section 3.1. Transitions, linking these states, are explained in detail in Section 3.2. As already mentioned, these two chapters do not compare the different tools, but give a list of all possible elements for state charts. In Section 3.3, we present a summarizing table, giving an overview of the possibilities and limitations of the notations analyzed in Section 3.1 and Section 3.2. In Section 3.4, we explain the meta-model, which we created from the analysis in the previous sections. And in the last section (i.e., Section 3.5), we discuss the *identity* of state charts elements. The identity defines how much the different model parts influence the functionality of the model and, thus, analyzing this property helps us to better understand the challenges during the adaption of the current approach to state charts.

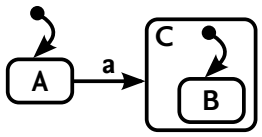
3.1. States

States describe the state of a system at a certain point in the execution. According to Harel [5], *states* are represented as rounded rectangles, which can encapsulate other states to express *hierarchy*. These states are connected with *transitions*, which are depicted by directed edges with an arrow at the end [5]. We explain these transitions in detail in Section 3.2 and concentrate on states for the moment. The *UML specification* defines the notion of *regions* [15]. Any state chart or state has a region, which contains all sub elements. Furthermore, the *UML specification* [15] defines, that stereotypes can be assigned to elements in state charts, which allow to define different roles for these elements. As *IBM Rational Rhapsody* is a tool realizing state machines according to the *UML specification* [15], it also provides the possibility to assign these stereotypes to states and transitions.

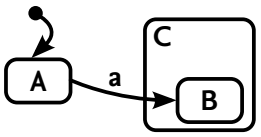
3.1.1. Initial states

States can be marked as *default states* (shown by a transition starting at a black bullet, going to the corresponding state [5]), showing which state is entered when the state chart, or a *hierarchy state* is entered. In Figure 3.3, we show an example for a state chart. It consists of two states `A` and `B`, which are connected by the transition `a`. In this example, `A` is defined to be the default state.

All four tools have different, but similar notions of default states. *The MathWorks Stateflow* and *IBM Rational Rhapsody* both use a transition, as Harel's state charts, to indicate that a state is a default state, except for they call these transitions a *default transition*. Both tools allow to set labels for these transitions, for example, to initialize variables at the start of execution. In *ETAS ASCET* and *Esterel Technologies SCADE Suite* the default state is marked by setting a corresponding flag on the state, which indicates that it is a default state. In *ETAS ASCET*, this kind of state is called a *start state*, and in *Esterel Technologies SCADE Suite*, it is an *initial state*. In the *UML specification* [5], default states are called *initial pseudostates*. A *pseudostate* in the UML is used to combine and direct transitions [15], and thus, does not represent a "normal" state with any functionality. In the course of this thesis, we will use the term initial state.



(a) Hierarchical state with a default transition



(b) Hierarchical state without a default transition

Figure 3.4.: Hierarchical states

3.1.2. Hierarchy States and Sub State Charts

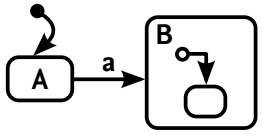
In Figure 3.4, we show an example for hierarchy states, as they are introduced by Harel [5]. The state chart consists of a state `A` and one hierarchy state `C`, containing a state `B`. In Figure 3.4a, `A` is defined to be the initial state for the state chart, but we also have to define a initial state inside the hierarchy state `C` (i.e., in this example state `B`), because otherwise, we do not know which state we should enter during the execution, when entering state `C`. Another way of defining, which state should be entered when entering a hierarchy state is a direct transition, crossing the bounds of the hierarchy state. In Figure 3.4b, a direct transition from state `A` to state `B` inside of state `C` exists, showing that this state is executed when entering the hierarchy.

In the *UML specification* these states are called *composite states*. In *Esterel Technologies SCADE Suite*, hierarchy states are called *hierarchical decomposition* and are realized by adding *state machines* to a state instead of directly adding sub-states to the state.

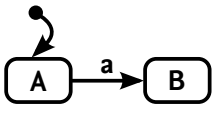
Harel [5] also discusses *unclustering*, which allows to define for any hierarchy state, that its contents are only shown in another view. This allows to reduce the complexity of state charts and improves the readability. In the *UML specification* [15], this kind of notion is called *submachine state*, in *The Mathworks Stateflow* it is called a *subchart*, and in *IBM Rational Rhapsody* it is called a *sub-statechart*. In the course of this thesis, we will use the term *sub state chart*.

IBM Rational Rhapsody realizes this element, by marking such states with miniature state chart pictograms, which indicate that this state contains such a sub state chart. In Figure 3.5, we present an example for a hierarchy state with a sub state chart. The state `B` in Figure 3.5a contains a sub state chart, indicated by the miniature pictogram of an initial state pointing to a state. When the user double-clicks on this state another window opens and presents the contents of the sub state chart. In Figure 3.5b, we show the contents of the corresponding sub state chart. In *ETAS ASCET*, sub state charts are realized indirectly, because the developer does not have to directly add them, but every hierarchy state is automatically realized as such a sub state chart, which is opened with a double-click.

This notion is only "syntactic sugar" for hierarchy states, in order to reduce the complexity when reading state charts, because hierarchy states directly display their contents (cf., Figure 3.4a) and

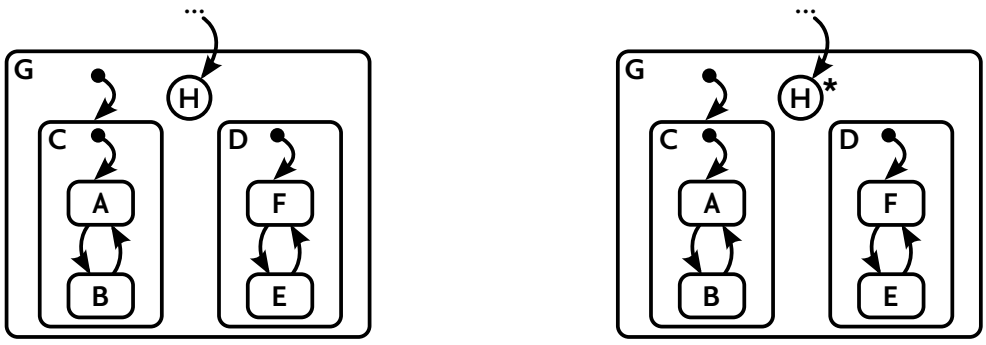


(a) Hierarchical state with a sub state chart pictogram



(b) Contents of the sub state chart

Figure 3.5.: Sub state charts



(a) Shallow history (b) Deep history

Figure 3.6.: Shallow and deep history

sub state charts only display their contents, when the user double-clicks on them (cf., Figure 3.5). Consequently, the sub state chart notion does not add any additional functionality compared to hierarchy states, because both notions allow to model hierarchy and the only difference is the way state charts are displayed.

3.1.3. History States

When a hierarchy state is reentered (i.e., after leaving the hierarchy state and entering it again afterwards), normally the initial state defined for the hierarchy state is reentered. However, sometimes it is convenient to reenter the state that was left, when leaving the hierarchy state. For this purpose Harel [5] introduces *history states* in two forms. *Shallow history* is indicated by an `H` in a solid circle and only remembers the history for the level, where it is defined. *Deep history* on the other hand, is also indicated by an `H` in a solid circle, but with a star (i.e., a *) beside it and remembers the history all the way down to the lowest level in the hierarchy. For example, in Figure 3.6a, the history state `G` only remembers, whether it was left from state `C` or state `D` and not which state inside them was last visited. In contrast, in Figure 3.6b the history state `G` would not only remember whether it was left from state `C` or state `D`, but also which inner state (i.e., state `A`, state `B`, state `E`, or state `F`) was last visited and reenters the corresponding state.

The terms shallow history and deep history were not introduced by Harel [5], but are part of the *UML specification* for state machines [15]. The deep history operator is only "syntactic sugar" for adding the history manually to every hierarchy level. All of the considered tools, except for *IBM Rational Rhapsody* and the *UML specification* [15], only support shallow history. Besides, the terms differ slightly. In *The MathWorks Stateflow* the history element is called *history junction*, in *ETAS ASCET*, it is only a *history* flag on a state, and in *IBM Rational Rhapsody* and *Esterel Technologies SCADE Suite*, it is called a *history connector*. The *UML specification* defines the term *history pseudostate* [15]. In the course of this thesis we will use the term history state.

3.1.4. Parallel States

The previously described states only allow to model *XOR states*. In other words, during the execution only one state could be active at a time. However, sometimes it is necessary to model *concurrent* execution and to allow state charts to be in more than one state at a time. According to Harel [5], this *orthogonality* is modeled by using hierarchy states and separating the parallel executed states

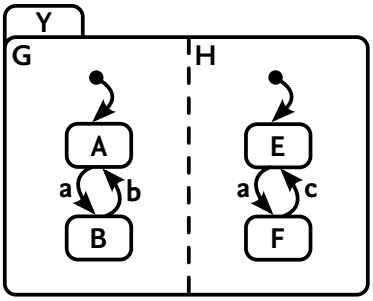


Figure 3.7.: Parallel states

by a dashed line [5]. As concurrency allows a state chart to be in more than one state at a time, the corresponding states are also called *AND states*. An AND state does not necessarily need to have a name assigned to it [5]. In this context, the *UML specification* uses multiple regions to separate the concurrent parts of the parallel state [15]. In Figure 3.7, we show an example for such an AND state. The AND state `Y` contains two regions `G` and `H` with the states `A`, `B`, `E`, and `F`. When executing such an AND state, the system has to be in all of the regions. Consequently, the initial state for the example in Figure 3.7, is `(A,E)`. During the execution, the parallel executed regions can change their state *simultaneously* (e.g., in Figure 3.7, when a change from `(A,E)` to `(B,F)` occurs) or independently (e.g., in Figure 3.7, when a change from `(B,F)` to `(B,E)` occurs). In the *UML specification* [15], these parallel states are called *orthogonal composite states*, in *IBM Rational Rhapsody*, they are called *orthogonal states*, in *The MathWorks Stateflow*, they are called *parallel states*, and in *Esterel Technologies SCADE Suite*, they are realized by adding multiple state machines to a state and consequently are called *parallel state machines*. In the course of this thesis, we will use the term parallel state.

3.1.5. State Actions

States can execute different kinds of actions. Harel [5] introduces *entry actions*, *exit actions*, and *throughout actions*. Entry actions are executed, when a state is entered via an incoming transition and exit actions are executed, when a state is left via an outgoing transition. Throughout actions are executed continuously while the system is in the corresponding state.

The MathWorks Stateflow calls throughout actions in their context a *during action* and also defines two additional actions. The *bind action* binds variables or events to the corresponding state. Afterwards, only this state or any of the state's children can change the bound variable or emit the bound event. The *on action* allows to set different types of *temporal actions*, which allow to execute the action after some temporal condition is fulfilled. This action type has two parameters. The first parameter is some positive integer `n`, which defines, according to one of the four temporal operators used, what happens when the `event` in the second parameter occurs. The *after operator* defines that the action should only be executed after the `event` occurred at least `n` times. The *before operator* defines that the action should only be executed before the `event` occurred `n` times. The *every operator* defines that the action should only be executed when the `event` occurred for the `n`-th time. And finally, the *at operator* defines that the action should only be executed at every `n`-th occurrence of the `event`.

In *ETAS ASCET*, throughout actions are called *static actions*, and *IBM Rational Rhapsody* uses *action on entry* and *action on exit* instead of entry action and exit action, respectively. The *UML specification* uses labels to define the actions for states and uses *entry*, *exit*, and *do* for entry actions, exit actions, and throughout actions, respectively [15]. Regarding state actions, *Esterel Technologies SCADE Suite* is

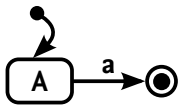


Figure 3.8.: Final states

a special case, because it defines its state actions with block-based models, as we described them in Section 2.1. In the course of this thesis, we will use the terms entry action, exit action, during action, bind action, and on action.

3.1.6. Final States

The *UML specification* [15], *IBM Rational Rhapsody*, and *Esterel Technologies SCADE Suite* introduce the notion of *final states* to state charts, which was not defined by Harel [5], but is used in classical automaton theory, where it is called *accepting state* [8, 18]. When such an accepting state is visited, the execution of the state chart is terminated, and this state cannot be left with any other transition. In Figure 3.8, we present an example for a final state. After taking the only transition from state `A`, we reach the accepting final state, which is indicated with a filled circle with another circle around it. The *UML specification* [15] and *Esterel Technologies SCADE Suite* both use the term of final states, whereas in *IBM Rational Rhapsody* they are called *termination states*. In *Esterel Technologies SCADE Suite*, a final state is not an extra element that can be added to the state chart, but normal states can be marked as final states by adding another edge around the state. In the course of this thesis, we will use the term final state.

3.2. Transitions

Transitions link the system states with each other and are depicted by directed edges with an arrow at the end. Transitions always have a *source state* and a *target state*, which consequently prohibits dangling edges (i.e., edges without a source state or a target state). In Figure 3.3, we can see such an arrow connecting state `A` with state `B`. Transitions not only connect states with other states, but can also create so-called *self loops* connecting a state with itself. In Figure 3.9, we can see such a self loop connecting state `B` with itself.

3.2.1. Transition Labels

In contrast to connections in block-based models, transitions can have a label with different properties. Harel [5] introduces *transition labels* of the form $\alpha(P)/S$. α represents an event that has to occur, in order to take the corresponding transition. P is some condition, that has to be fulfilled, otherwise the transition is not taken, even if the corresponding event occurred (i.e., both, the event and the condition, have to be fulfilled in order to take the transition). And the last value S of the label represents an action, which is executed when the transition is taken. In Figure 3.10, we present an example for transition labels. As we can see, the transition from state `A` to state `B` is only taken, if the event `ev` occurs and the condition `a == 0` is met. In this case, `a = 1` is executed.

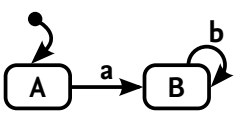


Figure 3.9.: Self loops

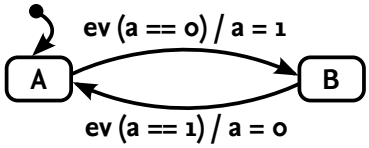


Figure 3.10.: Transition labels

Except for different wording the *UML specification* [15], *ETAS ASCET*, *IBM Rational Rhapsody*, and *Esterel Technologies SCADE Suite* use the same transition labels as Harel [5]. The *UML specification* [15], *ETAS ASCET*, *IBM Rational Rhapsody*, and *Esterel Technologies SCADE Suite* all use the word *trigger* instead of event. Besides, *IBM Rational Rhapsody* and the *UML specification* [15] both use *guard* instead of condition and *Esterel Technologies SCADE Suite* uses *trigger condition*. Instead of the term action, the *UML specification* [15] uses *behavior-expressions*. In the course of this thesis, we will use the terms event, condition, and action.

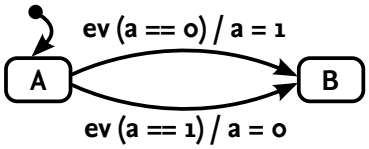
The MathWorks Stateflow distinguishes between two different types of actions for transitions. *Transition actions* are normal actions as defined by Harel [5], which are executed after the source state is left and before the target state is entered. The other action type are *condition actions*, which are executed as soon as the transition's condition is evaluated as `true`.

Nejati et al. [14] allow to assign multiple labels to transitions by using OR-operators. In Figure 3.11a, we present a state chart with two transitions from state `A` to state `B`. These two transitions can be merged into one transition by combining the two labels with an OR-operator (cf., Figure 3.11b). As we can see, this notion is only "syntactic sugar" for creating multiple transitions.

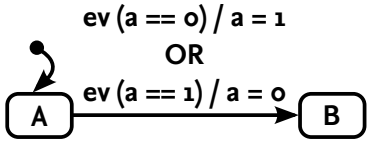
3.2.2. Conditionals and Selections

In order to simplify complex transitions, two notations exist. Harel [5] introduces *conditionals*, which are indicated by a `C` in a solid circle. This conditional allows to summarize transitions with the same event, but differing conditions. Consequently, we can use this notion for *if-then-else* conditions. In Figure 3.12, we can see how a conditional can improve the readability of such if-then-else conditions. In this example, the event `ev` is moved from all the outgoing transitions from state `D` in Figure 3.12a to only one outgoing transition from state `D` in Figure 3.12b. This outgoing transition enters a conditional, which has outgoing transitions for all conditions from the transitions in Figure 3.12b and represents the if-then-else part. The functionality of both state charts in Figure 3.12 is the same, but overall the readability and complexity is reduced by omitting duplicate information and reducing the number of transitions.

In *The MathWorks Stateflow* conditionals are called *connective junctions*, in *ETAS ASCET*, they are called *junctions*, in *IBM Rational Rhapsody*, they are called *condition connectors*, in *Esterel Technologies*



(a) Without multiple transition labels



(b) With multiple transition labels

Figure 3.11.: Multiple transition labels

(a) Without conditional (b) With conditional

Figure 3.12.: Conditionals

SCADE Suite, they are called *fork transitions*, and in the *UML specification* [15], they are called *choice* and are realized as pseudostates. In the course of this thesis, we will use the term conditional.

Besides conditionals, Harel [5] also introduces *selections*, which allow to enter states in a event-to-state mapping. In Figure 3.13a, we see an example, where the developer decided to model a set of clearly defined selection options as states. Consequently, selecting one of the options corresponds to emitting the event "`xy` selected", where `xy` is one of the options (i.e., in this example `A` or `B`). The selection notation (i.e., a transition going to an `S` in a circle) in Figure 3.13b allows to omit the corresponding transitions and automatically selects the states `A` and `B`, mapping the corresponding events when they are emitted. Thus, the total number of transitions in the state chart is reduced and the readability is improved.

Both notations are "syntactic sugar" for state charts, as they do not influence the functionality, but only allow to reduce the complexity of state charts.

3.2.3. Merges

Harel [5] introduces a means to organize transitions going into a state, in order to reduce the complexity and improve the readability, but does not present a name for this notion. The *UML specification* [15] uses the term *junctions* and realizes this element as a pseudostate. *IBM Rational Rhapsody* uses the term *junction connector*. *The MathWorks Stateflow* and *ETAS ASCET* both use the same terms for their notion of conditionals. In Figure 3.14a, we present an example without junction connectors. As we can see, state `C` has two incoming transitions with differing events `ev1` and `ev2`.

(a) Without selection (b) With selection

Figure 3.13.: Selections

(a) Without merge (b) With merge

Figure 3.14.: Merges

With a growing number of transitions this might get confusing, which is solved by the junction connector (cf., Figure 3.14b), because it joins a number of transitions going to the same state, before entering the corresponding state. As we can see, the junction connector is just "syntactic sugar" for the notation with multiple transitions going into one state and only reduces complexity without affecting the functionality. In the course of this thesis, we will use the term *merge*, which we introduce, because it best reflects the functionality of this notion.

3.2.4. Transition Priorities

All tools, except for *IBM Rational Rhapsody* and the *UML specification* [15], allow to assign *priorities* to transitions. These priorities extend Harel's [5] notion and are integers added to the transitions going out from a state and assign an execution order to the transitions. During the execution, every transition is checked in the order of the numbers, and the first transition is taken, whose events and conditions are fulfilled. In Figure 3.15, we present a state chart with priorities assigned to the transitions. If, for example, the event a occurred, the execution order would still be in the order of the priorities. So, in this example, the last checked transition will be taken, because it is the transition with the event a assigned to it.

3.2.5. Forks and Joins

The *UML specification* [15] and *IBM Rational Rhapsody* introduce *forks* and *joins* to split or join transitions. The notion of a fork should not be mistaken with fork transitions in *Esterel Technologies SCADE Suite*, which represent conditionals. A fork allows to split a transition in two or more outgoing transitions, which then enter different regions in a parallel state. Accordingly, a join merges two or more incoming transitions coming from different regions in a parallel state. In Figure 3.16, we present an example for forks and joins. The transition coming from state `A` is split into two transitions going into state `B` and state `C`, respectively. The outgoing transitions from state `B` and state `C` are joined together in one transition going in to state `D`. In the *UML specification* [15], these elements are realized as pseudostates.

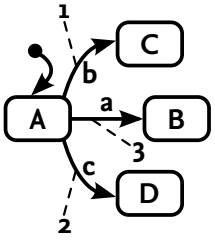


Figure 3.15.: Transition priorities

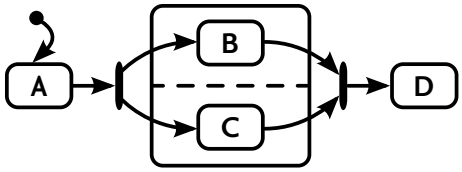


Figure 3.16.: Forks and Joins

3.2.6. Weak, Strong, Synchro, and Resuming Transitions

Regarding the way transitions behave, *Esterel Technologies SCADE Suite* is a special case, because it distinguishes between *weak transitions* (indicated with a blue bullet at the end of the transition) and *strong transitions* (indicated with a red bullet at the start of the transition). This distinction only affects the behavior during execution after the corresponding transition condition becomes `true`, as a strong transition directly activates the target state and does not execute the source state action. In contrast, a weak transition delays the activation of the target state by one cycle and first executes the action of the source state.

Beside, these two transition types, *Esterel Technologies SCADE Suite* also introduces *synchro transitions* (indicated with an `H*` in a circle at the end of the transition) and *resuming transitions* (indicated with a green triangle at the start of the transition and a blue bullet at the end of the transition). A synchro transition delays the execution of the target state until all final states from all parallel executions in a parallel state are finished.

Resuming transitions on the other hand, remember the inner execution state of a state, when it is left, and reenter it, when the state is reactivated. This notion is very similar to history states, and the only difference is, that we assign the history element to a transition, which allows to reenter the previous state only when entering the state via a certain transition. In Figure 3.17, we present an example for such a resuming transition. If the state `C` is left and is reentered via the transition from state `A`, it continues in the previous execution. If, on the other hand, the state `C` is reentered via the transition from state `B`, the execution of state `C` starts in its initial state.

3.2.7. Termination Connectors

Another element introduced by the *UML specification* [15] and *IBM Rational Rhapsody*, is the *termination connector*, which terminates the execution of the state chart and deletes the corresponding instance. In Figure 3.18, we can see the termination connector, which is marked with an `X`. In the *UML specification* [15], this element is called *termination pseudostate*. In the course of this thesis, we will use the term termination connector.

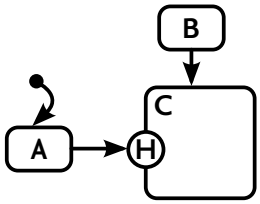


Figure 3.17.: Resuming transitions

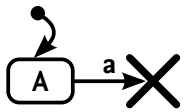


Figure 3.18.: Termination connectors

3.2.8. Diagram Connectors

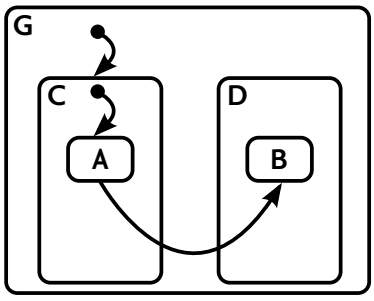
IBM Rational Rhapsody uses *diagram connectors*, which allow to improve the readability and to reduce transitions crossing other transitions or the boundaries of states (cf., Figure 3.19a). In order to use this element, we need to use two diagram connectors, which have the same name. In Figure 3.19b, we present such an example. Here, the two diagram connectors with the name `D` will connect. As we can see diagram connectors are only "syntactic sugar" for transitions, in order to reduce the number of crossed boundaries or transitions.

3.2.9. Spontaneous Transitions

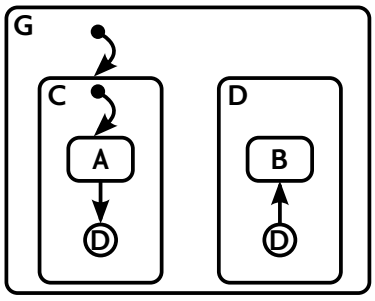
All notions presented allow to use *spontaneous transitions*, which means that these transitions do not have any events or conditions, which have to be fulfilled in order to take the corresponding transition. Consequently, these transitions are directly taken when the source state is entered. In Figure 3.20, we present such a spontaneous transition. Upon entering state `A` directly the transition to state `B` is taken (after any actions defined in state `A` are finished), because all prerequisites are fulfilled.

3.2.10. Enter Points and Exit Points

Two elements defined by the *UML specification* [15] and *IBM Rational Rhapsody* are *enter points* and *exit points*. Both elements are used for submachine states or hierarchy states, in order to avoid transitions crossing the edges of these states. In Figure 3.21a, we present an example, where transitions cross the edges of the state `C`. This is eliminated in Figure 3.21b, by using enter points and exit points. As we can see, enter points and exit points are only "syntactic sugar" for transitions crossing the boundaries of states. In the *UML specification* [15], these elements are realized as pseudostates.



(a) Without diagram connector



(b) With diagram connector

Figure 3.19.: Diagram connectors

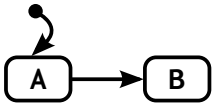
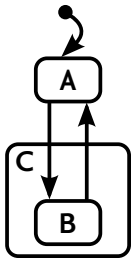
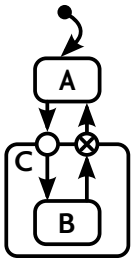


Figure 3.20.: Spontaneous transitions



(a) Without Enter Points and Exit Points



(b) With Enter Points and Exit Points

Figure 3.21.: Enter Points and Exit Points

3.3. Overview of the analyzed State Chart Notations

For the purpose of comparing the possibilities and limitations of the notations analyzed in Section 3.1 and Section 3.2, we created Table 3.1, giving an overview over their language elements. In the left column, we show the identified elements. For each notation, we created a column, showing the name for the corresponding element if it is supported by the notation, but its name differs from the other notations. If the names are the same for all notations, we only show, whether the corresponding notation supports them or not.

3.4. Meta-model

In this section, we describe the meta-model, which we will use in the course of this thesis to create instances of state charts and compare them with each other. Since our goal is to use our family mining approach for different tools, we use the results of the previous analysis to create the meta-model.

In Figure 3.22, we show a part of the whole class diagram for the meta-model. This part only shows how state charts and their states are modeled, omitting transitions, their actions, and actions for states. As we can see, the root element for a state chart is the `StateChart` class, which stores the total number of states, regions, and transitions contained in the corresponding state chart (i.e., `stateCount`, `regionCount`, and `transitionCount`). All different state types extend the abstract class `State`, which allows to assign stereotypes to these states. We distinguish between the concrete classes `InitialState`, `NormalState`, `FinalState`, and `TerminationConnector`. Since final states and termination connectors both end the execution of a state chart (i.e., they are *end states*), the corresponding classes inherit from the abstract class `EndState`. This allows to treat both types of end states in the same manner, if desired. The `TerminationConnector` identified during the analysis is a special transition, which can be represented by a `Transition` going to a `State` of type `TerminationConnector`. We could have realized all state types by using only one class for states with different boolean flags, indicating what kind of state the corresponding instance represents. We decided against such a solution, because we would have needed further logic to ensure, that a

	Harel's state charts [5]	The MathWorks Stateflow R2012b	ETAS ASCET 6.1.3	IBM Rational Rhapsody 8.0.6	Esterel Technologies SCADE Suite R15a	UML 2.4.1 [15]
State charts	State charts	State machines	State machines	State charts	State machines	State machines
Stereotypes	–	–	–	✓	–	✓
States						
Initial states	default state	default transition	start state	default transition	initial state	initial pseudostate
Hierarchy states	✓	✓	✓	✓	✓ (hierarchical decomposition)	✓ (composite state)
Sub state charts	unclustering	subchart	(✓)	sub-statechart	–	submachine state
History states	history	history junction	history	history connector	history connector	history pseudostate
Deep / shallow history	✓/✓	–/✓	–/✓	✓/✓	–/✓	✓/✓
Parallel states	orthogonality	parallel state	–	orthogonal state	parallel state machines	orthogonal composite state
State actions	entry action exit action throughout action	entry action exit action during action bind action on action	entry action exit action static action	action on entry action on exit	block-based models	entry exit do
Final states	–	–	–	termination state	final state	final state
Transitions						
Transition event	event	event	trigger	trigger	trigger	trigger
Transition condition	condition	condition	condition	guard	trigger condition	guard
Transition actions	action	condition action transition action	action	action	action	behavior-expression
Conditionals	conditional	connective junction	junction	condition connector	fork transition	choice pseudostate
Merges	✓ (no name)	connective junction	junction	junction connector	–	junction pseudostate
Selections	✓	–	–	–	–	–
Transition priorities	–	✓	✓	–	✓	–
Forks and Joins	–	–	–	✓	–	✓ (pseudostates)
Termination connectors	–	–	–	termination connector	–	termination pseudostate
Diagram connectors	–	–	–	✓	–	–
Spontaneous transitions	✓	✓	✓	✓	✓	✓
Enter Points and Exit Points	–	–	–	✓	–	✓ (pseudostates)
Weak / strong transitions	–	–	–	–	✓	–
Synchro / resuming transitions	–	–	–	–	✓	–

Table 3.1.: Overview over the tools

state is only one of the corresponding types. By using a more complex inheritance hierarchy (cf., Figure 3.22), we do not need such validation to ensure mutual exclusion between the different types.

Every of the concrete state classes implements at least one of the two interfaces `IncomingState` and `OutgoingState`. These two interfaces determine, whether the corresponding state can have incoming or outgoing transitions. Consequently, the `NormalState` class implements both interfaces, since it can have both, incoming and outgoing transitions. The `InitialState` class only implements the `OutgoingState` interface, because initial states are the start of the execution, and therefore, only can have outgoing transitions. Correspondingly, `EndStates` only implement the `IncomingState` interface, because they end the execution and cannot have any outgoing transitions.

Each `StateChart` has a root region, which is an instance of the `Region` class. The `Region` class stores at least one normal state, and, if they exist initial states and end states. The `Region` class was introduced to the meta-model, in order to allow to model parallel states with multiple regions. Besides allowing to model parallel states, these regions enable us to create states with hierarchy by adding only one region to the corresponding state. Only normal states can be hierarchy states or parallel states, thus, only the `NormalState` class allows us to add sub-regions. In order to allow navigation in both directions, each region also has a `NormalState` as a parent. The only case where this parent can be `null` is when the corresponding region is the root region of the state chart. According to the regions, each state also has a parent to define the region it belongs to. As states can only be added to regions, this parent can never be `null`. The `State` class has two methods `isHierarchical()` and `isParallel()`, which check for the corresponding state, whether it is a normal state. When this precondition is met, the methods check whether the corresponding normal state is a hierarchy state (i.e., it contains exactly one subregion), or a parallel state (i.e., it contains more than one subregion).

In Subsection 3.1.2, we also introduced sub state charts, and argued that they are only "syntactic sugar" for hierarchy states, because both notions display hierarchy, and only their visual representation is different. Thus, we do not model sub state charts separately and `NormalStates` are the only class, which allows to define history, because the history operator is not sensible for non-hierarchical states as initial states or end states. The history is modeled by using the `HistoryOperator` enum, which has the three values `NONE` (the default value), `SHALLOW`, and `DEEP` to represent states without history, with shallow, and with deep history.

In Figure 3.23, we present the part of the class diagram, showing how the different state action types are realized. As we can see, only normal states can hold state actions, because initial states and end states only start, or end the execution of the state chart and consequently do not execute any actions. All state actions inherit from the abstract `Action` class, which allows to store the code for the corresponding action. Since we can also have actions for transitions, we added the abstract classes `StateAction` and `LabelAction` as another hierarchy level to distinguish between these two types. In the lowest hierarchy level, we have the classes for the different state actions, which inherit from `StateAction`. These classes are `EntryAction`, `ExitAction`, `DuringAction`, `BindAction`, and `OnAction`. `OnActions` are a special case, because they define that actions are executed according to some temporal operator and a number of occurrences (i.e., the `temporalValue` variable). We use the `TemporalOperator` enum with the `AFTER`, `BEFORE`, `EVERY`, and `AT` operators to specify when the action should be executed.

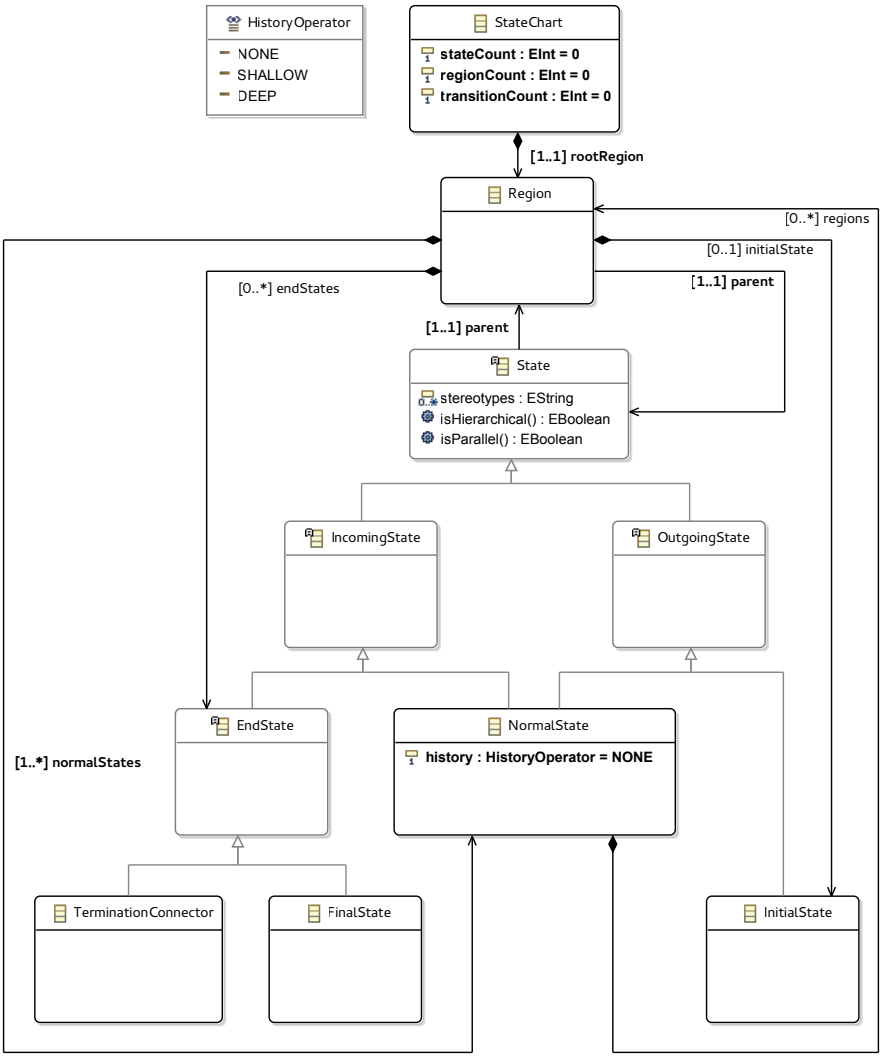


Figure 3.22.: Class diagram for the states

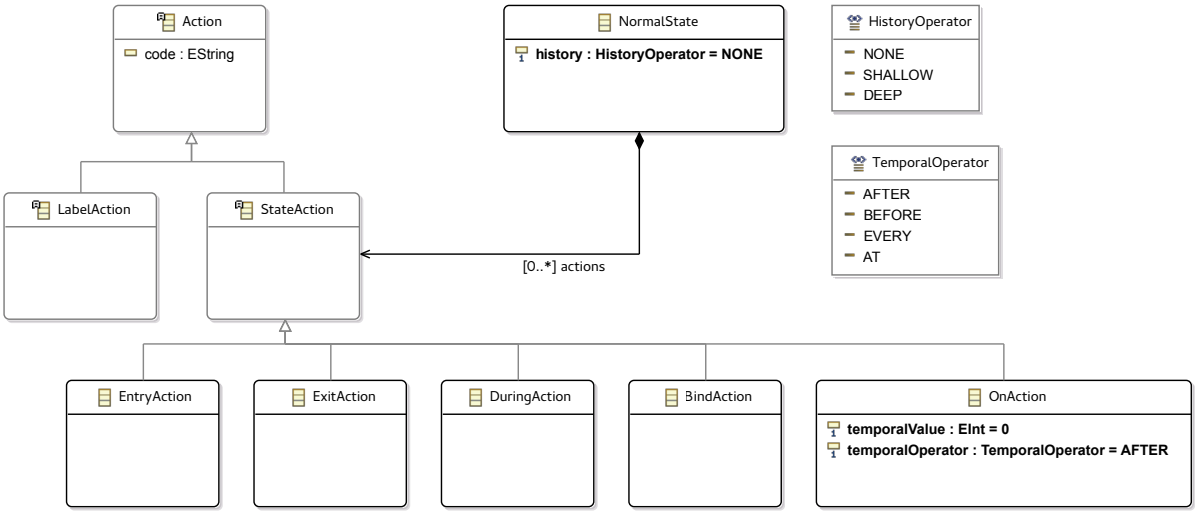


Figure 3.23.: Class diagram for the state actions

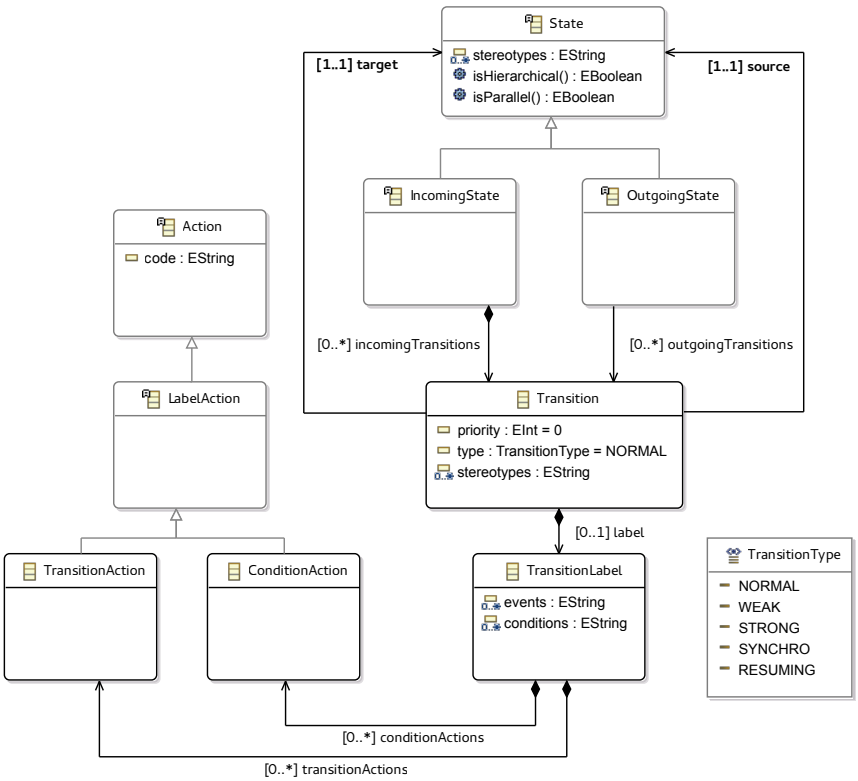


Figure 3.24.: Class diagram for the transitions

In Figure 3.24, we present the part of the class diagram, which shows the details of transitions and their transition labels. As we can see, a `Transition` always has a source and a target state. These states implement the `IncomingState` and `OutgoingState` interfaces, and, consequently, hold a list of incoming transitions, outgoing transitions, or both if they implement both interfaces. A transition can have a priority, stereotypes, and a transition type. This type is realized by using the `TransitionType` enum with the five values `NORMAL` (the default value), `WEAK`, `STRONG`, `SYNCHRO`, and `RESUMING`.

Each transition can hold a transition label. If this label is not set, the transition represents a spontaneous transition. Such a transition label has events and conditions. In addition, it can hold `TransitionActions` and `ConditionActions`. These action types both inherit from `LabelAction`, in order to be able to process both types in the same manner.

As we explained in Subsection 3.2.5, Forks and Joins are used to split or join transitions, that are entering or leaving different regions of a parallel state. Since, these two elements do not directly influence the functionality, but only show, that the transition going out from a state enters or leaves the regions of a parallel state simultaneously, they are only "syntactic sugar". They can easily be represented by multiple transitions leaving from a state and entering the regions, or the other way round. Thus, we do not use separate elements for forks and joins. The other elements introduced as "syntactic sugar" in Section 3.2 are not modeled, because they do not add any additional information for the family mining approach to the meta-model, since they do not influence the functionality, and only allow to reduce the complexity when visualizing state charts. Consequently, conditionals, selections, merges, diagram connectors, and enter and exit points are not modeled in the meta-model, as they can be modeled using simple `Transitions`.

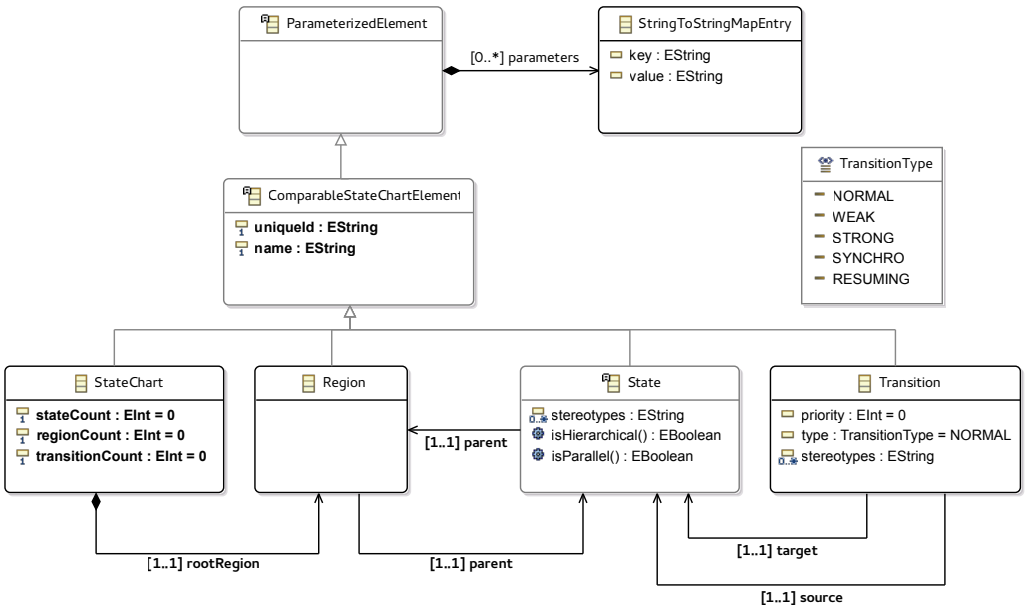


Figure 3.25.: Class diagram for further elements

In Figure 3.25, we present some further elements of the meta-model, which we introduce in order to process state charts more easily. All elements, which we want to compare (i.e., `States`, `Transitions`, `Regions`, and `StateCharts`) implement the `ComparableStateChartElement` interface, which allows to assign a name and a unique id (e.g., a *universally unique identifier (UUID)*) to these elements. The `ComparableStateChartElement` interface, in turn, implements the `ParameterizedElement` class, which allows to annotate elements by adding parameters to a `Map`, which maps `Strings` to `Strings`.

In Table 3.2, we summarize, how all identified language elements, of the different tools, can be represented using the classes and attributes of the meta-model described in this section. In the left column, we can see the identified elements from the analyzed notations, and in the right column, we summarize the classes and attributes representing them in the meta-model. As we can see, all important elements identified during the analysis are represented in the meta-model. Consequently, the created meta-model can be used to create instances of state charts for the different compared notations, and, thus, allows to apply family mining to them using suitable algorithms.

3.5. Identity of State Chart Elements

Before we can adapt our current approach for family mining of block-based models (cf., Section 2.4) to state charts, we reason about the *identity* of the elements in both model types. The identity of an element defines, which parts influence the functionality of a model, and which of these properties are important, when comparing two elements. Properties influencing the functionality of an element are, for example, the block types of blocks in block-based models, because they define how the corresponding block behaves during execution. On the other hand, properties such as the block's name or its color, are not influencing the functionality. By analyzing the identity of state chart elements, we better understand, which properties need to be considered during the comparison of different elements. Besides, we identify challenges, which might hinder the adaption of the current approach, and how we can tackle them, in order to successfully apply family mining to state charts.

Identified element	Meta-model representation
State charts	`StateChart`
Stereotypes	`stereotypes` field in `States` and `Transitions`
States	`State`
Initial states	`InitialState`
Hierarchy states	`NormalState` with one `Region` element
Sub state charts	`NormalState` with one `Region` element
History states	`history` field for `HistoryOperator` in `NormalState`
Deep / shallow history	`HistoryOperator` enum values `DEEP` and `SHALLOW`
Parallel states	`NormalState` with more than one `Region` element
State actions	`StateActions`: `EntryAction`, `ExitAction`, `DuringAction`, `BindAction`, and `OnAction`
Final states	`FinalState` (inherits from `EndState`)
Transitions	`Transition` and `TransitionLabel`
Transition event	`events` field in `TransitionLabel`
Transition condition	`conditions` field in `TransitionLabel`
Transition actions	`TransitionActions`: `ConditionAction` and `TransitionAction`
Conditionals	multiple `Transitions`
Merges	multiple `Transitions`
Selections	multiple `Transitions`
Transition priorities	`priority` field in `Transitions`
Forks and Joins	multiple `Transitions`
Termination connectors	`Transition` to `TerminationConnector` (inherits from `EndState`)
Diagram connectors	`Transition`
Spontaneous transitions	`Transition` without `TransitionLabel`
Enter Points and Exit Points	`Transition` to `State` in `Region`
Weak / strong transitions	`type` field in `Transition` with `TransitionType` enum values `WEAK` and `STRONG`
Synchro / resuming transitions	`type` field in `Transition` with `TransitionType` enum values `SYNCHRO` and `RESUMING`

Table 3.2.: Overview over the meta-model classes

3.5.1. Identity of Elements in Block-based Models

For the family mining of block-based models, we mainly consider the properties of the blocks contained in the models and their neighborhood. Connectors are only considered indirectly during the comparison of blocks and neighbors, as we follow them and identify the blocks' successors and predecessors (only when comparing the neighborhood). Since connectors for block-based models only show the data flow and do not influence the functionality / behavior of the models, they are not represented in the used similarity metric. The properties considered for blocks during family mining of block-based models are summarized in Table 3.3. As we can see, the identity of a block is defined by three properties, which are not directly influencing the block's functionality (i.e., name, inports, and outports), and one property, which defines its behavior during the execution (i.e., the block type). The name has very low impact on the similarity metric of two compared blocks, since it does not influence the functionality of the block, and it does not need to be unique. Besides, it can contain spelling mistakes, which might distort the results during comparison. The similarity metric of the inports and outports is calculated by comparing the neighbors connected to the block. For example, when comparing the inports of two blocks, we compare the number of similar predecessors of the compared blocks by comparing their names and functions. Two predecessors are

Property	Influences the functionality	Impact on similarity metric
name	–	very low
block type	✓	very high
inports	(✓)	low
outports	(✓)	low

Table 3.3.: Properties defining the identity of blocks in block-based models

considered similar, if their name and block type are the same. By calculating the average of similar predecessors with respect to the total number of predecessors, we identify the similarity metric of the block's neighborhood. The same method applies for the outports of blocks, except that we compare the successors.

As the neighborhood of compared blocks defines their execution environment, the inports and outports have an indirect influence on the blocks' functionality and a higher impact on the similarity metric of the blocks than their names. Consequently, we assign a slightly higher impact to these properties. The only properties for blocks in block-based models, which have direct influence on the block's functionality are the block type and any associated parameters. Thus, they have very high impact on the similarity metric during the comparison of two blocks, since they define how the block behaves during execution. In addition, the names of block types are unique for most block-based languages, since most of these languages provide a library with elements that can be used, and have unique names. For example, in *The MathWorks MATLAB/Simulink* a library exists, which contains different block types such as `Sum`, `Integrator`, and `Product` blocks. These block names are unique for the whole library and consequently very easy to compare.

3.5.2. Identity of States in State Charts

Defining the identity of state charts is more complex, since not only the states (i.e., the equivalent for blocks in block-based models), but also the transitions (i.e., the equivalent for connectors in block-based models) have to be considered, because their labels also define parts of the state chart's functionality, since they can trigger events with their actions.

In Table 3.4, we present a table with all important properties of states. As we can see, we consider eleven properties for states. Similar to the identity of blocks in block-based models, the name of a state has very low impact on its behavior, since it does not influence the functionality, does not need to be unique and can contain spelling mistakes. The two properties, whether a state is a start state or an end state do not influence the execution in such a way, that they have a direct influence on the functionality, because they only influence where the execution starts or ends, but not how the system behaves (cf., state actions). Consequently, start states and end states have a low impact on the similarity metric of two states. Parallel states only allow to create multiple regions for a state, which are executed in parallel. As the functionality of the corresponding state is defined by its sub-states and sub-transitions, we assign a very low impact on the similarity metric to this property, because it only influences the execution indirectly.

The *hierarchy distance* between two states has no influence on the functionality, because it only shows, whether two states are on the same *hierarchy level*, and in this case, how many hierarchy

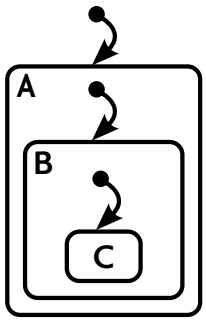


Figure 3.26.: Example for hierarchy levels

levels exist in between. The hierarchy level of a state is defined by the number of hierarchical parents, where the root region of a state chart has hierarchy level zero. With every added hierarchical state the hierarchy level is increased. In Figure 3.26, we present an example for hierarchy levels. In this example, `state A` is on hierarchy level zero, since it is directly contained in the root region. Consequently, `state B` is on hierarchy level one and `state C` is on hierarchy level two. With these hierarchy levels, we can calculate the hierarchy distance between two compared states by using Equation 3.1.

$$h_d = |h_l(s_1) - h_l(s_2)| \tag{3.1}$$

This equation takes uses the function h_l, which returns the hierarchy level of the compared states s_1 and s_2, and calculates the absolute difference between these hierarchy levels. The result (i.e., h_d) shows the distance between the compared states, where zero means, that they are on the same level and, consequently, are more likely to be a reasonable comparison. Every value greater than zero shows, how many hierarchy levels are between the two states. For example, comparing the two states `state A` and `state C` in Figure 3.26 would result in $h_d = 2$. As the hierarchy level only shows how likely it is, that two states are similar according to their hierarchy level, we assign a low impact on the similarity metric to this property.

Similar to block-based models, the neighbors of states have only indirect influence on the functionality of a state chart and, thus, have a limited impact on the similarity metric. When comparing the neighbors of states, we apply the same algorithm as for block-based models, but have to compare the names and the actions of the states, instead of block types. The stereotypes of states have no influence on the functionality and consequently their impact on the similarity metric of compared states is low.

The history of hierarchical states influences the functionality of the corresponding states, because it changes the execution when the hierarchy is left and re-entered later, as it remembers the last executed state and re-enters it. Consequently, the execution is altered and the impact of this property on the similarity metric is high. Every state (except for initial states) is *dependent on events*, since the state is only executed when some event occurs and it is triggered by this event. Consequently, we compare the transition labels of all transitions entering the compared states. As this property influences the execution strongly, it has very high impact on the similarity metric of compared states.

The main functionality of a state is defined by its actions, which are executed when the state is entered and the preconditions for the corresponding actions are fulfilled. Hence, the *triggered*

Property	Influences the functionality	Impact on similarity metric
name	–	very low
start state	(✓)	low
end state	(✓)	low
parallel state	(✓)	very low
hierarchy distance	–	low
neighbors	(✓)	low
stereotype	–	low
history state	✓	high
dependent on events	✓	very high
triggered actions	✓	very high
events triggering change	✓	low

Table 3.4.: Properties defining the identity of states in state charts

actions of a state strongly influence the functionality and have a very high impact on the similarity metric of compared states, because they can change variables and can trigger new events. Every state (except for end states) has *events triggering changes*, since states can have outgoing transitions, which are triggered when some event occurs. Consequently, these events are triggering a change of state. As these transitions are also incoming transitions for other states, these events are also considered during the comparison of these states, when comparing the property *dependency on events*. Thus, we assign a lower impact on the similarity metric to them, although they strongly influence the functionality.

3.5.3. Identity of Transitions in State Charts

In Table 3.5, we present a table with all important properties of transitions. Similar to states, the name has a very low impact on the similarity metric of two compared transitions, since it does not influence the functionality, does not need to be unique and can contain spelling mistakes. The stereotypes of transitions do not have any influence on the functionality and, consequently, their impact on the similarity metric of compared transitions is low. The type of the transition (i.e., weak, strong, resuming, synchro) influences the way a transition is executed and, thus, has a higher impact on the similarity metric of compared transitions, but still does not influence the functionality as much as events, conditions, or the transitions' actions. The priority of transitions controls their execution order and, thus, influences the functionality. Similar to transition types, it does not influence the functionality as much as events, conditions, or the transitions' actions. Consequently, it has a low impact on the similarity metric of compared transitions.

Events and conditions strongly influence the execution of state charts, as events trigger state changes and conditions influence when transition are executed (including their actions and entering a triggered state). Hence, these two properties have high impact on the similarity metric of compared transitions. The actions executed by the transitions (transition actions and condition actions) have the highest influence on the functionality, because these actions can trigger new events and change variables and states. Thus, we rate the impact of these properties on the similarity metric of compared transitions very high.

Property	Influences the functionality	Impact on similarity metric
name	–	very low
stereotype	–	low
type	✓	low
priority	✓	low
events	✓	high
conditions	✓	high
condition actions	✓	very high
transition actions	✓	very high

Table 3.5.: Properties defining the identity of transitions in state charts

As we can see, adapting the current approach needs to consider, states and transitions, in order to produce reasonable results. In addition, more properties have to be considered to compare state charts, because compared to our current approach for block-based models many more elements contribute to the functionality of a model.

3.5.4. Creating a Concrete Metric

The analysis of the impact of the described properties for states and transitions on the corresponding similarity metrics allows us to define a *concrete metric* for the comparison of state chart elements. A concrete metric defines weights (i.e., a value between 0.0 and 1.0), which assign different importance to the considered properties. For example, the name of a state gets a lower weight than the triggered events property, because it has a lower impact on the state's functionality and, thus, on the corresponding similarity metric. We distinguish between two main categories for properties, *static properties* and *dynamic properties*. Static properties do not have any impact on the functionality of the corresponding element (e.g., the name does not influence the functionality), whereas dynamic properties influence the behavior during the execution and, thus, the functionality of the element (e.g., the triggered events property influences the behavior of the state).

In Equation 3.2, we present the calculation of the overall similarity s of two elements e_1 and e_2 by considering the elements' static properties sp and dynamic properties dp. This equation needs a set of weights W which consists of three subsets $W_{sp} \subsetneq W$, $W_{dp} \subsetneq W$, $W_o \subsetneq W$. W_{sp} and W_{dp} define the weights for the static and dynamic properties of the compared elements. W_o defines the overall weights applied to the sums of weighted static and dynamic properties.

$$s(e_1, e_2) = w_s \cdot \sum_{i=0}^{m-1} w_{sp_i} \cdot s_{sp_i}(e_1, e_2) + w_d \cdot \sum_{j=0}^{n-1} w_{dp_j} \cdot s_{dp_j}(e_1, e_2) \tag{3.2}$$

The weight for static properties $w_s \in W_o$ and the weight for dynamic properties $w_d \in W_o$ allow us to assign different importance to the overall similarity of these properties. The functions s_{sp_i} and s_{dp_j} calculate the similarity of the elements' static property sp_i and dynamic property dp_j, respectively. Such a function should define an algorithm to compare the corresponding properties of two elements e_1 and e_2 (e.g., for two state names a simple string comparison for equality could be sufficient) and return a similarity value between 0.0 and 1.0 in order to show their similarity. These similarities are multiplied with the corresponding weights $w_{sp_i} \in W_{sp}$ and $w_{dp_j} \in W_{dp}$, which are

defined by the similarity metric. After summing up all weighted similarities for the static and dynamic properties, these sums are multiplied with the corresponding weights $w_s \in W_o$ and $w_d \in W_o$, resulting in the final similarity s of the compared elements.

For example, when comparing two elements a and b with two static properties x and y and one dynamic property z, we compare the properties of the elements with each other and calculate their similarity (e.g., $s_{sp_0} = 0.5$ for x, $s_{sp_1} = 0.25$ for y, and $s_{dp_0} = 0.75$ for z) and multiply them with the corresponding weights (e.g., $w_{sp_0} = 0.25$, $w_{sp_1} = 0.75$, and $w_{dp_0} = 1$). After summing up the weighted similarities of static properties (i.e., $0.5 \cdot 0.25 + 0.25 \cdot 0.75 = 0.3125$ for our example) and dynamic properties (i.e., $0.75 \cdot 1 = 0.75$ for our example), we multiply them with the corresponding weights (e.g., $w_s = 0.25$, $w_d = 0.75$) and know the final similarity s of the compared elements (i.e., $s(a,b) = 0.3125 \cdot 0.25 + 0.75 \cdot 0.75 = 0.640625 \approx 64.06\%$).

Important to notice is, that the sums of all weight subsets (i.e., $W_{sp} \subsetneq W$, $W_{dp} \subsetneq W$, and $W_o \subsetneq W$) should always be exactly 1.0, because otherwise these subsets would not use the full range to weight the different properties (i.e., if the sum is less than 1.0), or would produce similarities greater than 100% (i.e., if the sum is greater than 1.0).

In this section, we do not introduce a concrete metric for state charts, because we argue that concrete metrics are highly dependent on the used state chart types. For example, *IBM Rational Rhapsody* allows to use stereotypes, which cannot be used in *The Mathworks Stateflow*. Consequently, a corresponding metric should consider such circumstances and should only comprise properties available in the corresponding tools. In Chapter 6, we introduce such a concrete metric, which we use to evaluate our implementation of family mining for state charts with *IBM Rational Rhapsody* state charts from a case study.

3.6. Summary

In this chapter, we analyzed the state chart notations by Harel [5], *The MathWorks Stateflow*, *ETAS ASCET*, *IBM Rational Rhapsody*, *Esterel Technologies SCADE Suite*, and the *UML specification* [15], in order to identify, what kind of language elements they introduce to model state charts. In addition, we presented a table, which allows to compare the different notations and to identify, which tool uses which language elements, and how they are called in each tool. Furthermore, we presented a meta-model for state charts, which can be used to transfer state charts, created with the analyzed notations, into our internal representation and apply family mining to them. Finally, we analyzed the identity of elements in block-based models and state charts. This identity defines how much impact elements have on the functionality of the corresponding model and how much attention needs to be paid to these elements, when comparing the models.

4 Approach

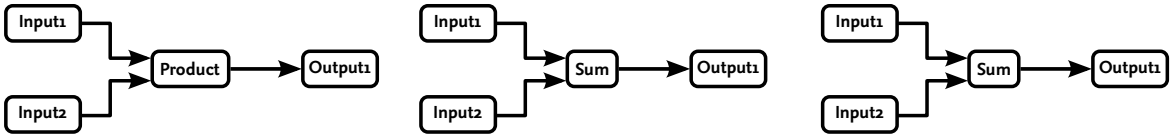
In this chapter, we discuss all challenges, which need to be tackled in order to successfully adapt the phases of the current algorithm for block-based models to apply family mining to state charts and, furthermore, present the solutions, which we utilize to solve them. During the analysis of the existing workflow for block-based models, we identified an issue regarding the way we process multiple models. In Section 2.4, we describe our current workflow (in the following referred to as the OldWorkflow), where we select one base model and compare all other models with this base model. The results of these comparisons are merged into one final 150% model. In Subsection 2.3.1, we describe the workflow for our previous approach [6], which selects one base model and compares it with one of the other models. The results are merged back into a 150% model, which is the input for the next comparison with another model. In Section 4.1, we explain in detail why this workflow will create better results than the OldWorkflow and argue that the existing workflow should be refactored. Keeping the ideas of this refactored workflow (in the following referred to as the NewWorkflow) in mind, we discuss in the following sections, how the phases of the current family mining algorithm for block-based models can be adapted to state charts.

In Section 4.2, we explain, how we can adapt the *Comparing Phase* of the current approach for block-based models (cf., Subsection 2.4.1), in order to apply family mining to state charts. This entails to adapt the comparison of blocks to states and find a way, how transitions and regions can be compared. In Section 4.3, we explain the adaption of the current *Matching Phase* (cf., Subsection 2.4.2) to the created compare elements. And finally, we explain how the matched compare elements can be merged into one model, by adapting the current *Merging Phase* (cf., Subsection 2.4.3). In Section 4.5, we explain the challenges during the adaption of the family model exporter from the current approach. And finally, in Section 4.6, we introduce a new idea for a family mining algorithm, which takes advantage of the characteristics of state charts to compare them with each other.

4.1. Refactoring the Workflow

In Figure 2.9 in Section 2.4, we show the OldWorkflow for family mining for block-based models. As we can see, we select a base model from the set of models, that should be compared. All other models are defined as compare models. In the *Compare Phase*, each of these compare models is compared with the selected base model. The resulting lists of possible matches are distinctively matched in the *Matching Phase*, and in the *Merging Phase* the resulting lists of distinct matches are merged one after another into the base model to create a 150% model for the compared models. The resulting 150% model is either used as another input to merge the next model, or is exported as a family model.

The described OldWorkflow only compares the compare models with the selected base model and merges the results into a final 150% model. The results of the OldWorkflow might be imprecise, because certain situations are not identified correctly. We use the three models in Figure 4.1 to discuss one possible issue, which is related with the OldWorkflow. As we can see,



(a) Model 1 (b) Model 2 (c) Model 3

Figure 4.1.: Example for possible problems with the OldWorkflow

Model 3 is a copy of Model 2 and Model 1 only differs from these two models because of its Product block instead of the Sum block. Using the current family mining approach for block-based models and assuming that Model 1 is selected as the base model, we create two lists of distinct matches for the three models. For the comparison between Model 1 and Model 2, this list is the same as for the comparison between Model 1 and Model 3: (Input1,Input1), (Input2,Input2), (Product,Sum), and (Output1,Output1).

During the *Merging Phase*, the three compare elements (Input1,Input1), (Input2,Input2), and (Output1,Output1) from both comparisons are easy to process, because, according to the calculated similarity, they should be the same and, consequently, should be identified as mandatory blocks. The problematic compare elements are the (Product,Sum) compare elements from both comparisons. When exporting the family model, after merging the identified variability from the comparison of Model 1 and Model 2, the resulting family model would look like in Figure 4.2a. As we can see, the alternative blocks Sum and Product are correctly identified as such. In the following merging step, the corresponding 150% model is used as an input to merge the results from the comparison of Model 1 with Model 3 into the final 150% model. One would expect, that the Sum block from Model 3 is identified as the same block as the Sum block from Model 2, which was previously identified to be alternative to the Product block from Model 1. Consequently, there should not be any change to the created 150% model from the first merging step. However, we compared both compare models, Model 2 and Model 3, with the base model Model 1 and cannot identify, that the two Sum blocks are the same, because both compare elements are (Product, Sum). Thus, after the second merging step the 150% model would contain another alternative Sum block. In Figure 4.2b, we present the resulting family model.

In order to prevent this undesired behavior, we could change the order of the compared models and use, for example, Model 2 as the base model. All compare elements for the two compare models

(a) Family Model for Model 1 and Model 2 (b) Family Model for Model 1, Model 2, and Model 3

Figure 4.2.: Family models for the compared models in Figure 4.1

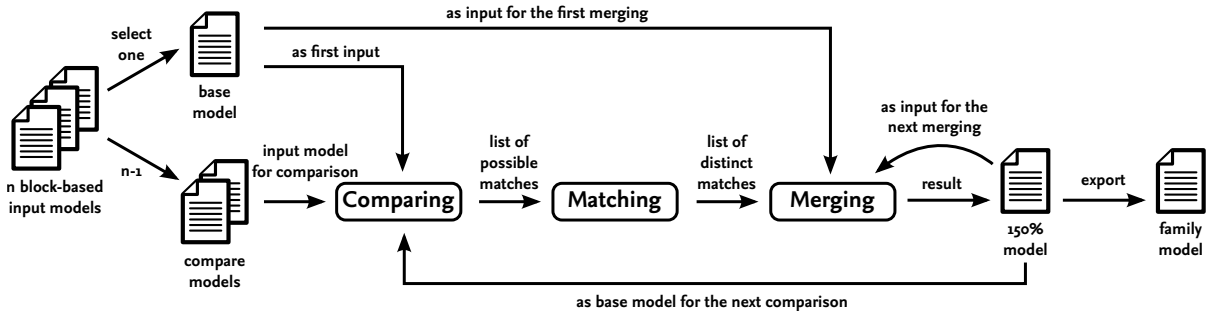


Figure 4.3.: NewWorkflow for family mining for state charts

`Model 1` and `Model 3` would be the same, except for the compare elements `(Sum, Product)` and `(Sum,Sum)`, respectively. These two compare elements allow us to correctly identify the variability between the `Sum` and the `Product` block, and also show that the `Sum` block from `Model 2` is the same as the block from `Model 3`. In this case, the resulting family model would meet the expectations and would look like in Figure 4.2a.

Using this approach, the results of family mining would be very non-deterministic, since the results would be highly dependent on the order of the processed models. This behavior is not desirable, because the results would vary a lot. Consequently, we refactor the workflow of the current approach for family mining for block-based models during the adaption for state charts and argue that the existing OldWorkflow should also be modified to overcome the presented issue. In Figure 4.3, we present the refactored NewWorkflow, which is similar to the workflow presented in [6]. As we can see, the selected base model is only used as an initial input for the comparison with the first compare model and furthermore, for the first merging step. Afterwards, the created 150% model is used for the comparison with the next compare model and also is the basis for the next merging step. For each compared model, the NewWorkflow walks through all phases (i.e., the *Comparing Phase*, the *Matching Phase*, and the *Merging Phase*) of the family mining approach. This cycle is repeated until all models were compared and merged into a final 150% model.

For our example, we would first compare `Model 1` with `Model 2` and identify the corresponding variability. We present the resulting family model in Figure 4.2a. Afterwards, the 150% model, which was used to export the family model in Figure 4.2a is the input for the next comparison. The created compare elements after the comparing step are: `(!In1,In1)`, `(!In1,In2)`, `(!In2,In1)`, `(!In2,In2)`, `(⇔Product,Sum)`, `(⇔Sum,Sum)`, and `(!Out1,Out1)`. As we can see, these compare elements contain exclamation marks (i.e., mandatory elements), arrows (i.e., alternative elements), and question marks (i.e., optional elements), showing information about the previously identified variability. For example, `Product` and `Sum` were previously identified as alternatives to each other. Consequently, new compare elements contain the corresponding variability information (i.e., in this case the ⇔) for these elements and this information has to be considered during the processing of new compare elements.

In case of mandatory or optional elements inside a new compare element, we do not have to consider this information until we start the *Merging Phase*, where we have to decide, whether the existing variability has to be changed according to the new compare element. For example, an existing mandatory element can become optional, if it is not contained in a new compared variant of a model.

In case of alternative elements, we also have to consider previously assigned variability information during the *Comparing Phase* and *Matching Phase*. When comparing a new element with an element, which was previously identified to be alternative, we also have to consider the information, that it is part of an *alternative group*. In block-based models, we introduced a hierarchical block grouping and containing the alternative blocks, which is called a `VariantSubsystem`. In order to reduce the number of hierarchical elements in compared state charts, we annotate alternative groups with a number. All alternatives, which belong to the same group have the same group number (e.g., the alternatives `Sum` and `Product` would have the same group number). When comparing another state with such a group, we have to create all possible compare elements for the states from the alternative group compared with the new state (i.e., in our example `(⇔Product,Sum)` and `(⇔Sum,Sum)`). Since there can only be one match per element, we have to decide, which element from the alternative group matches best with the new element. During the *Merging Phase*, this decision can either result in a new alternative element in the existing alternative group, or no changes, when the new element is equal to one of the existing elements in the alternative group. For our example, the best match would be `(⇔Sum,Sum)`. Consequently, no new alternative has to be created during the *Merging Phase*, because the two `Sum` blocks are the same. Thus, the 150% model from the previous *Merging Phase* is not modified (i.e., the resulting family model is Figure 4.2a).

In this section, we argue that the OldWorkflow for family mining for state charts (and also for block-based models) should be refactored (cf., Figure 4.3) to solve the presented problem. During the adaption of the OldWorkflow to the new NewWorkflow, we have to consider the presented ideas:

- For each compared model, we have to walk through the whole family mining process: *Comparing Phase*, *Matching Phase*, and *Merging Phase*.
- The base model is *only* used as an initial input for the *Comparing Phase* and the *Merging Phase*. When comparing more than two models, the results of the previous step are used as new inputs for the next *Comparing Phase* and for the next *Merging Phase*.
- When comparing a `Block A` with another `Block B`, which was previously identified to be alternative to some other blocks, we have to compare `Block A` with `Block B` *and* all its alternatives. From these compare elements, we have to select the best compare element, since there can only be one distinct match for a block with such an alternative group. The selected compare element either represents a new alternative to all existing alternative elements, or an element, which is equal to some existing element and, thus, does not have to be added the alternative group. In case of family mining for state charts, we have to apply the same ideas during the comparison of transitions.

4.2. Adapting the Comparing Phase

For the adaption of the current *Comparing Phase*, it is important to find a way to consider all relevant elements in state charts, which have an identity and, thus, have impact on their functionality. In Section 3.5, we discussed this identity and now have to consider this information to compare states and transitions with each other. In case of states, many concepts from the current approach can be adapted, since blocks and states have a similar structure, except for their properties and certain

(a) `Model 1` (b) `Model 2`

Figure 4.4.: State charts, compared to explain the adaption

differences in their structure (e.g., blocks use sub-blocks to model hierarchy, whereas regions are used as an extra layer to model sub-states). In addition, we have to identify an easy way to integrate the comparison of transitions into the current approach, in order to compare all relevant elements used in state charts.

When adapting the *Compare Phase* from the current approach for block-based models, we use the same algorithm to create the compare elements. We select a *base state chart* from the list of all input state charts and use all other state charts as *compare state charts*. These are the equivalents for base models and compare models from the current approach for block-based models. In Figure 4.4, we present two state charts, which we compare, in order to explain the adaption of the current approach. In this example, we assume that `Model 1` is selected as the base state chart. For the base state chart and all compare state charts, we find the *initial states* and use them as the start for our comparison (i.e., in `Model 1`, we find `initial state`, and in `Model 2`, we find `initial state`). These initial states are the equivalents for the lists of start states from the current approach for block-based models. In contrast to the current approach for block-based models, we only have a single state as start point, since state charts can only have one start point. Consequently, the creation of the first compare element only compares the found initial states of the models, which are currently compared (i.e., `(initial state,initial state)` in our example).

The current approach for block-based models now creates a list of all subsequent blocks by following the outgoing connectors and storing the connected target blocks in lists. These lists are then used to create new compare elements by comparing each list element from the base model with all list elements from the compare model. This step is repeated until no new compare elements are created. As the transitions in state charts also have an impact on their functionality, we also have to compare transitions with each other. We could have compared all states in the state charts first, and afterwards, we could have started another comparison for all transitions. Instead, we decided to compare states and transitions in one run of the compare algorithm, in order to be more efficient, as we only iterate once over the models. Thus, the next step during the comparison of state charts is to create the lists of all subsequent transitions for the initial states (i.e., transition `initial transition` in `Model 1` and transition `initial transition` in `Model 2`), instead of all subsequent states.

Now, the lists of all subsequent transitions are compared with each other (i.e., the compare element `(initial transition,initial transition)` is created in our example), and only then we create the lists of all subsequent states from these transitions (i.e., `state A` in `Model 1` and `state D` in `Model 2`) and compare them with each other (i.e., compare element `(A,D)` is created for our example). Important to notice is that, the lists used for subsequent states and transitions are represented by two disjunct lists, in order to better distinguish between the elements, which are compared. Afterwards, again the lists of subsequent transitions are created (i.e., transition `a` in `Model 1` and

(a) Non-hierarchical `Block A`

(b) Non-hierarchical `Block B`

Figure 4.5.: Comparing two non-hierarchical blocks

transition `c` in `Model 2`) and compared with each other (i.e., compare element `(a,c)` is created for our example). This step is repeated until no new compare elements are created (i.e., after creating the compare elements, `(B,E)`, `(b,d)` and `(C,F)` we stop). As we can see, the basic adaption of the current algorithm for block-based models to compare state chart elements with each other is easy if we do not use any hierarchical states or parallel states in state charts, since we can directly compare states and transitions with each other. In order to prevent the algorithm from running into an infinite loop (e.g, a state has a self-loop or a transition back to a state, which was previously processed), we check for each iteration, whether the found elements were already considered during the comparison. If not, the algorithm continues with the creation of new compare elements. Otherwise, it has to stop, in order to prevent an infinite loop.

4.2.1. Comparing Hierarchical Block-based Models

Similar to hierarchical blocks in block-based models, comparing two states with a hierarchical region or parallel regions is more complex. For block-based models, we distinguish between four different comparison types for blocks:

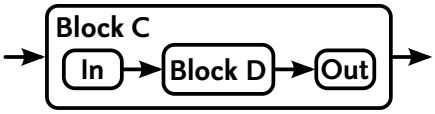
1. A non-hierarchical block from the base model is compared with a non-hierarchical block from the compare model (cf. Figure 4.5).

   The first comparison type is already explained in Subsection 2.4.1, and we present an example in Figure 4.5. As we can see, it is easy to realize a comparison of `Block A` with `Block B`, since we only have to compare the properties of the blocks, which are listed in Table 3.3.

2. A non-hierarchical block from the base model is compared with a hierarchical block from the compare model, or the other way round (cf. Figure 4.6).

   For the second comparison type, we only compare the blocks' names and their neighborhood, since we cannot compare the block type of a non-hierarchical block with the block type of a hierarchical block. In Figure 4.6, we present such an example. As we can see, the names and interfaces of `Block A` and `Block C` can easily be compared. The functionality of these blocks cannot be compared, since their block type differs (e.g., in *The Mathworks MATLAB/Simulink* we would compare the block type of `Block A` with the block type `Subsystem` of `Block C`).

(a) Non-hierarchical `Block A`



(b) Hierarchical `Block C`

Figure 4.6.: Comparing a non-hierarchical block with a hierarchical block

(a) Hierarchical `Block A` (b) Hierarchical `Block C`

Figure 4.7.: Comparing two hierarchical blocks

3. A hierarchical block from the base model is compared with a hierarchical block from the compare model (cf. Figure 4.7).

   For the third comparison type, we compare the blocks' names and interfaces, but do not compare their block type, as it is the same for both blocks. Instead, we compare the contents of the hierarchical blocks by recursively starting the compare algorithm for these blocks. Afterwards, we also run the matching algorithm (cf., Subsection 2.4.2) for the created compare elements and add the matched results to a list in their parent compare element. In Figure 4.7, we present such an example. As we can see, we can easily compare the names and interfaces of `Block A` and `Block C`. For the blocks contained in these hierarchical blocks, we start the compare algorithm again and compare `In`, `Block B`, and `Out` with `In`, `Block D`, and `Out` from the other model. The created compare elements `(In,In)`, `(Block B,Block D)`, and `(Out,Out)` are then matched and added as sub compare elements to the compare element (`Block A`,`Block C`).

4. One of the compared blocks is `null`:

   a) The base model block is compared with `null`.

   b) The compare model block is compared with `null`.

   The fourth comparison type is very easy to handle, since we only have to create compare elements, which compare the block from the base model or the compare model with `null` (i.e., the compare elements would be `(block,null)` and `(null,block)`, respectively).

4.2.2. Comparing Hierarchical State Charts

In order to apply family mining to state charts, we have to adapt the algorithms described in Subsection 4.2.1 in order to compare hierarchical and parallel states with each other. Besides, we have to develop algorithms to compare transitions with each other, since they are an important part of the functionality in state charts (cf., Section 3.5) and are not considered in block-based models.

Comparing States In order to adapt the algorithms described in Subsection 4.2.1 for hierarchical and parallel states, we have to add further comparison types. Consequently, we have the following list of comparison types:

1. A non-hierarchical and non-parallel state from the base state chart is compared with a non-hierarchical and non-parallel state from the compare state chart (cf., Figure 4.8).

   Similar to the first comparison type for block-based models, the comparison of two non-hierarchical and non-parallel states can easily be adapted from the current approach, since the way the properties of the two element types are compared, is basically the same. The corresponding properties are listed in Table 3.4. In Figure 4.8, we present an example for two

(a) Non-hierarchical and non-parallel `State A` (b) Non-hierarchical and non-parallel `State B`

Figure 4.8.: Comparing two non-hierarchical and non-parallel states

non-hierarchical states. As we can see, we can compare their properties and interfaces without further effort and are able to directly create the corresponding compare element.

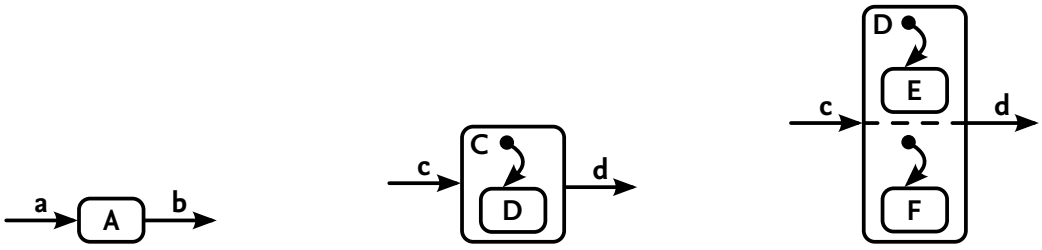
2. A non-hierarchical and non-parallel state from the base state chart is compared with a hierarchical or parallel state from the compare state chart, or the other way round (cf., Figure 4.9).

   Similar to the second comparison type for block-based models, we can only compare the name, interface, and in addition all properties, which do not directly influence the functionality of the state, when we compare a non-hierarchical and non-parallel state with a hierarchical or parallel state. All properties influencing the functionality (i.e., dependent on events, triggered actions, events triggering change, and history state) is defined by the sub-states of the hierarchical or parallel state and not by their actions.

   In Figure 4.9, we present such an example. As we can see, we can adapt the current approach for block-based models by comparing only the name, the *interface*, and the properties, which do not influence the functionality of the state (e.g., stereotypes and the hierarchy distance), when we compare the state in Figure 4.9a with Figure 4.9b or Figure 4.9c. During the comparison of the states, the interfaces are compared by comparing their neighborhood, meaning, that we compare the states' direct predecessors and successors regarding their name and their actions. By calculating the average of predecessors and successors, which are the same according to these properties, we receive the similarity of the states' interfaces.

3. A hierarchical state from the base state chart is compared with a hierarchical state from the compare state chart (cf., Figure 4.10).

   Similar to the third comparison type for block-based models, we can use a recursive approach to compare hierarchical states. In Figure 4.10, we present two hierarchical states, which we can compare by comparing all properties, which do not directly influence their functionality (e.g., name, interface, and hierarchy distance). All properties influencing the functionality (i.e., dependent on events, triggered actions, events triggering change, and history state) are



(a) Non-hierarchical and non-parallel `State A` (b) Hierarchical `State C` (c) Parallel `State D`

Figure 4.9.: Comparing a non-hierarchical state with a hierarchical or parallel state

(a) Hierarchical `State A` (b) Hierarchical `State C`

Figure 4.10.: Comparing two hierarchical states

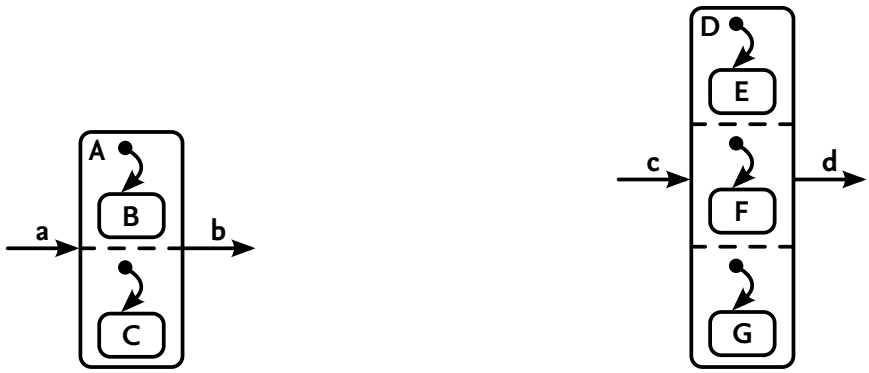
defined by the sub-states of the hierarchical state and not by their actions. Consequently, we start a new compare run for the sub-regions of `State A` and `State C` and compare their sub-states (i.e., `initial state`, `State B` and `initial state`, `State D`) and sub-transitions (i.e., `initial transition` and `initial transition`). Afterwards we run the matching algorithm on the created compare elements (i.e., `(initial state,initial state)`, `(State B,State D)`, and `(initial transition,initial transition)`). The resulting compare elements for states and transitions are stored in the created compare element for the compared regions of `State A` and `State C`, which is stored in the parent compare element `(State A,State B)`.

The similarity of a region compare element can be calculated by summing up all similarities of the contained sub compare elements for states and transitions and dividing them by the total number of contained elements. The new compare element for regions is not really necessary for the comparison of hierarchical states, because we could simply compare the sub-contents of these states and add the corresponding compare elements to the compare element of the compared hierarchical states. Since, we need the region compare element during the comparison of parallel states, in order to distinguish between the sub-contents of the different regions, we also use it for hierarchical states.

4. A parallel state from the base state chart is compared with a parallel state from the compare state chart (cf., Figure 4.11).

   The fourth comparison type for state charts uses some of the ideas, which we apply in order to compare hierarchical states. A parallel state consists of multiple regions. Consequently, we can use the same approach as for hierarchical states and compare all properties, which do not directly influence the functionality of the parallel states (e.g., name, interface, and hierarchy distance), since their functionality is defined by their sub-states and sub-transitions. For the comparison of the sub-regions, we have to do some extra work, since we do not know, which combinations of the regions are the most similar. Nonetheless, the algorithm to compare two regions is the same as for hierarchical states (i.e., we compare all sub-contents and match them).

   In Figure 4.11, we present two parallel states `State A` and `State D`. As we can see, the parallel states consist of two regions for `State A` and three regions for `State D`, respectively. In order to find the best result during the comparison of these regions, we have to compare all possible combinations of the regions from `State A` and `State D`. For this example, we assume that the regions are named after the state contained in them. Consequently, the list of combinations is `(Region B, Region E)`, `(Region B,Region F)`, `(Region B,Region G)`, `(Region C, Region E)`, `(Region C,Region F)`, and `(Region C,Region G)`. In order to find the best combination, we can use the standard match algorithm, which only has to be



(a) Parallel `State A` (b) Parallel `State D`

Figure 4.11.: Comparing two parallel states

adapted to work for regions. Given that, the compare elements `(Region B,Region E)` and `(Region C,Region G)` are the best matches for the example in Figure 4.11, `Region F` is the only remaining region, since it was not matched with another region. This region has to be compared with `null`, because it represents an optional region.

5. A hierarchical state from the base state chart is compared with a parallel state from the compare state chart, or the other way round (cf., Figure 4.12).

   The fifth comparison type for state charts, is very similar to the comparison of two parallel states, because we compare a hierarchical state (i.e., with one region) with a parallel state (i.e., with more than one region). Consequently, we can apply the same idea and create all possible combinations from the found regions and run the adapted match algorithm to find the best combination. This algorithm will find the best compare element for the region from the hierarchical state. All remaining regions from the parallel state are treated as optional regions and, consequently, are compared with `null`. In Figure 4.12, we present an example for this comparison type. In this example, we have to create the compare elements `(Region B,Region E)` and `(Region B,Region F)` and find the best compare element. The remaining region, has to be treated as an optional element and is compared with `null`.

6. One of the compared states is `null`:
   a) The base state chart state is compared with `null`.
   b) The compare state chart state is compared with `null`.

(a) Hierarchical `State A` (b) Parallel `State D`

Figure 4.12.: Comparing a hierarchical state with a parallel state

(a) 150% model after comparing non-hierarchical and non-parallel `State A` and `State B`

(b) Non-hierarchical and non-parallel `State C`

Figure 4.13.: Comparing an alternative state with another state

The sixth comparison type can be handled similarly to the fourth comparison type for block-based models, since we only have to create compare elements, which compare the state from the base state chart or the compare state chart with `null` (i.e., the compare elements would be `(state,null)` and `(null,state)`, respectively).

7. More than two state charts are compared and a state, already marked as alternative, is compared with a state from a compare state chart.

   The seventh comparison type is introduced, because we introduced the NewWorkflow for family mining (cf., Section 4.1). When comparing more than two state charts, we have to consider states, which are already annotated with variability, and which are marked as alternative. During the comparison, all states from the corresponding alternative group have to be compared with the compared state and the best combination has to be selected.

   In Figure 4.13, we present such an example. In a previous comparison, `State A` and `State B` were identified to be alternative. When comparing `State C` with ⇔`State A`, we identify that `State A` is part of the alternative group with ⇔ `State B` and that we have to compare both states with `State C`. In order to prevent, comparing the same alternative group again with the same state, we have to remember, that we already processed this alternative group. For example, this could happen when `State A` and `State B` have the same mandatory predecessor. Consequently, we would create the lists of successors from this state (i.e., ⇔ `State A` and ⇔ `State B`), which we would have to compare with `State C`. Since ⇔ `State A` and ⇔ `State B` are part of the same alternative group, we would repeat the same comparisons for these states. For large alternative groups this would produce unnecessary overhead, which would slow down the overall algorithm. After creating the possible compare elements for the corresponding alternative group, we identify the best match for the comparison of the alternative group with the corresponding state (e.g., (⇔`State A,State C`)). The other compare elements (i.e., for this example (⇔`State B,State C`)) are eliminated from the list of possible matches, since they have a lower similarity than the best match.

Comparing Transitions The comparison of transitions with each other is much easier, since we do not have any hierarchical elements and can directly compare the transitions. Consequently, we distinguish only between the following two comparison types:

1. A transitions from the base state chart is compared with a transition from the compare state chart.

   For this comparison type, we can directly compare the properties of the transitions, which are listed in Table 3.5, and store the results in transition compare elements.

2. One of the compared transitions is `null`:
   a) The base state chart transition is compared with `null`.
   b) The compare state chart transition is compared with `null`.

   This comparison type can be handled similarly to the states, since we only have to create compare elements, which compare the transition from the base state chart or the compare state chart with `null` (i.e., the created compare elements would be `(transition,null)` and `(null,transition)`, respectively).

4.2.3. Pseudocode for the Adapted Comparing Phase

In Algorithm 4.1, we present the pseudocode for the adapted *Comparing Phase*. The compare algorithm needs two state charts sc_b and sc_c as input, which represent the base state chart and the compare state chart, respectively. In Line 2 and Line 3 two lists for the created possible state and transition compare elements (i.e., CE_{p_s} and CE_{p_t}) are initialized. Furthermore, in Line 4 and Line 5 two lists are initialized with the initial states from the base state chart ($initial_{b_s}$) and the compare state chart ($initial_{c_s}$) and afterwards the compare algorithm for states is started.

By calling the method in Line 8 the comparison of states is started. In the following, the algorithm checks, whether the two lists $succ_{b_s}$ and $succ_{c_s}$ passed to this method are empty. If both lists contain at least one state (cf. Line 9), all possible combinations for these states are created in Line 12. If one of the lists is empty, (cf. Line 15 or Line 19) the elements from the other non-empty list are compared with *null* (cf. Line 17 and Line 21). All created compare elements are stored in the list of possible compare elements for states (CE_{p_s}).

After comparing all states contained in the lists passed to the *compareStates*() method, the lists of successors for the states from the base state chart are created in Line 24 and for the states from the compare state chart Line 25. As explained in Section 4.2, the algorithm continues to compare the successor transitions for these states. Thus, the created lists contain transitions and if one of these lists is not empty, the *compareTransitions*() method is called in Line 27. In order to prevent comparison loops, the implementation of the successor determination methods should check, whether they were already processed in a previous comparison and, thus, should not be considered in following comparisons.

The *compareTransitions*() method body is only outlined, since it basically contains the same contents as the *compareStates*() method. It only differs insofar, that it compares transitions instead of states and that its created successor lists contain states instead of transitions. With these lists it calls the *compareStates*() method instead of the *compareTransitions*() method. Consequently, these two methods are called alternately as described in Section 4.2.

We do not provide further pseudocodes to explain the algorithms executed by the *compare*() methods, which are called by the *compareStates*() and *compareTransitions*() methods, since they can be realized by following the explanations and enumerations in Subsection 4.2.2. As the comparison of multiple regions is a little more complex, such that, we directly call the matching algorithm after creating the corresponding compare elements, we explain in Algorithm 4.2 how states with regions can be compared.

In this case, the two states $s_{b_{hp}}$ and $s_{c_{hp}}$, which are either two parallel states, or one hierarchical state and one parallel state, are passed to the method. The algorithm first initializes the list CE_{p_r} for the

Input: Two state charts $sc_b, sc_c \in SC, sc_b \neq sc_c$
Output: Lists CE_{p_s} and CE_{p_t} with all possible compare elements for states and transitions

```
1  method compare(sc_b, sc_c)
2      CE_ps ← ∅;
3      CE_pt ← ∅;
4      initial_bs ← initialState(sc_b);
5      initial_cs ← initialState(sc_c);
6      compareStates(initial_bs, initial_cs);
7  return;
8  method compareStates(succ_bs, succ_cs)
9      if |succ_bs| > 0 and |succ_cs| > 0 then
10         foreach s_b ∈ succ_bs do
11             foreach s_c ∈ succ_cs do
12                 CE_ps ← compare(s_b, s_c);
13             end
14         end
15     else if |succ_bs| > 0 then
16         foreach s_b ∈ succ_bs do
17             CE_ps ← compare(s_b, null);
18         end
19     else if |succ_cs| > 0 then
20         foreach s_c ∈ succ_cs do
21             CE_ps ← compare(null, s_c);
22         end
23     end
24     succ_bt ← succ(succ_bs);
25     succ_ct ← succ(succ_cs);
26     if |succ_bt| > 0 or |succ_ct| > 0 then
27         compareTransitions(succ_bt, succ_ct);
28     end
29 return;
30 method compareTransitions(succ_bt, succ_ct)
31     ...
32 return;
```

Algorithm 4.1.: Pseudocode for the adapted *Comparing Phase*

Input: Two parallel or hierarchical states $s_{b_{hp}}, s_{c_{hp}} \in S, s_{b_{hp}} \neq s_{c_{hp}}, s_{b_{hp}} \in sc_b, s_{c_{hp}} \in sc_c, sc_b \neq sc_c$
Output: List CE_{m_r} with all matched region compare elements

```
1  method compareHierarchicalParallelStates(s_{b_hp}, s_{c_hp})
2      CE_{p_r} ← ∅;
3      CE_{m_r} ← ∅;
4      regions_b ← regions(s_{b_hp});
5      regions_c ← regions(s_{c_hp});
6      foreach r_b ∈ regions_b do
7          foreach r_c ∈ regions_c do
8              CE_{p_r} ← compare(r_b, r_c);
9          end
10     end
11     CE_{m_r} ← match(CE_{p_r});
12 return CE_{m_r};
```

Algorithm 4.2.: Pseudocode for the comparison of regions

created possible region compare elements (cf. Line 2) and the list CE_{m_r} for their distinct matches (cf. Line 3). In Line 4 and Line 5 the lists of regions contained in the two compared states are copied to the two lists $regions_b$ and $regions_c$. Each of the region elements in these lists is compared with all region elements from the other list and the corresponding compare elements are stored in CE_{p_r} (cf. Line 8). As a last step these compare elements are matched in Line 11 and the corresponding results are stored in CE_{m_r} and are returned.

4.3. Adapting the Matching Phase

In order to find distinct matches for the compare elements created during the *Comparing Phase*, we have to adapt the *Matching Phase* from the current approach for block-based models (cf. Subsection 2.4.2). As we have three different compare element types (i.e., for states, transitions, and regions), we need to adapt the matching algorithm as generically as possible in order to reduce the overhead during the implementation of our approach. This matching algorithm iterates over the list of created compare elements and tries to find distinct matches for model elements. While processing the list of created compare elements, we check whether other compare elements exist that contain either the same element from the base model or the same element from the compare model. If no other compare element exists, we can directly match the two elements from the processed compare element. If other compare elements are found, which contain the same elements, we select the compare element with the highest similarity and delete all ruled out compare elements from the list of possible matches.

In some cases, we cannot directly match elements, because there exist other relevant compare elements with the same similarity value. Consequently, no distinct match is possible and we move these ambiguous elements to the end of the list and continue matching, hoping that other matches resolve these conflicts. If the conflict cannot be resolved and we revisit these ambiguous elements, we have to present these problematic compare elements to the developers, in order to get a manual

decision, which is the most appropriate match. After processing all compare elements and solving possible conflicts, there are certain cases, where we did not match every element with another element from the other model or `null`. For each of these elements without a matching partner a new compare element comparing the element with `null` is created.

The current algorithm only takes the calculated similarity of the created compare elements into account and does not access the properties of the compared elements (i.e., the compared blocks) to determine the distinct matches. It only needs access to these compared elements inside of the compare element, in order to check, whether better compare elements exist with the same base element or compare element. Consequently, we can adapt the match algorithm in a generic way, in order to treat all state chart elements (i.e., states, regions, and transitions) in the same manner and by applying the same matching algorithm to them.

4.3.1. Pseudocode for the Adapted Matching Phase

In Algorithm 4.3, we present a short excerpt from the algorithm used for the matching of all compare elements (i.e., for states, transitions, and regions). This algorithm is realized recursively and expects a list of possible compare elements. This list should only contain compare elements from a certain type (i.e., states, transitions, *or* regions) and not a mixture of different types. Otherwise, the matching algorithm does not create sensible results.

First the list of distinct matches CE_m is initialized in Line 2 and afterwards the matching algorithm is executed in Line 3 by calling the *matchNext*() method. The algorithm works directly on the sorted list of possible compare elements and uses *ambiguous* and *handled* flags to distinguish between compare elements, which are ambiguous (i.e., they cannot be matched distinctively, since other relevant compare elements with the same similarity exist) and compare elements, which were processed in a previous execution. When sorting the list, all compare elements without flags are sorted to the start of the list, followed by the *handled* elements, the *ambiguous* elements, and the elements, which are flagged *handled* and *ambiguous*.

If the first element x in the CE_p list is not marked with any flag, the match algorithm tries to distinctively match the corresponding compare element by calling the *tryToMatch*() method in Line 9. This method first creates the list SBE in Line 19 with all compare elements from the list CE_p, which contain the same base element as x. If the created list is empty and the element compared to the base element in x is *null* (cf. Line 20), x can directly be matched (cf. Line 21) and the matching algorithm is called in Line 22 for the next element in CE_p. Since distinctively matching a compare element x rules out all other compare elements containing the same base element b or compared element c, calling the *match*() method entails deleting the corresponding compare elements from CE_p, because they become invalid.

If the element from the compare state chart is not *null* (cf. Line 23), the algorithm has to check whether another compare element y exists with the same element c from the compare state chart, which has a higher similarity than x (cf. Line 24). In this case the *checkComparedElement*() method proceeds similar to the *tryToMatch*() method and first identifies all compare elements, which contain the same element c. If the corresponding list is empty, x can be directly matched. If there exists another element y in this list, which has a higher similarity than x, the algorithm marks x as *handled*, moves y to the start of CE_p, and calls *matchNext*() again. If only compare elements y with the same or a lower similarity as x exist, the algorithm marks all compare elements y and x,

which have the same similarity, as *ambiguous*. Afterwards, the algorithm sorts the list and calls the *matchNext*() method again. If only compare elements y with a lower similarity than x exist. In this case, the algorithm directly matches x and continues with the next element from CE_p by calling the *matchNext*() method again.

The last case, which has to be considered is when the list SBE contains elements (cf. Line 26). In this case, the algorithm first checks in Line 27 whether SBE contains any y with a higher similarity than x. In this case, the *hasBetterBaseElementMatch*() method is used to identify, if the corresponding compare element y is optimal, or if other compare elements exist with the same compared element, but a higher similarity. If y is identified as optimal it is directly matched and the algorithm tries to match another element by calling the *matchNext*() method in Line 28. If there is no better compare element y, the algorithm checks for compare elements y, which have the same similarity as x (cf. Line 30). In this case, the corresponding compare elements x and y are marked *ambiguous*, CE_p is sorted (cf. Line 31), and the algorithm continues with the next compare element (cf. Line 32). If there are no compare elements y, which have the same similarity as x, x might represent an optimal element and the algorithm only has to check in Line 34, whether any other compare element exists, which has the same compared element and a higher similarity. Afterwards, the algorithm continues with the next compare element in Line 35.

At the end of the execution there are no compare elements left in CE_p and all elements from the base state chart or compare state chart without a distinct match are processed in Line 15 by comparing them with *null*. For example, this occurs for a compare element x, which compares the states s_{b1} and s_{c1}, when another compare element y comparing the states s_{b2} and s_{c1} has a higher similarity. After matching y, the state s_{b1} does not have a matching partner and, thus, has to be matched with *null*.

If the first element in CE_p is marked as *ambiguous*, the algorithm was not able to distinctively match this element, since other elements exist, which have the same similarity. When such an element is identified by the *matchNext*() method in Line 6, all compare elements in CE_p are at least marked as *ambiguous* and the algorithms processes these elements by calling the *processOptimalAndAmbiguousElements*() method in Line 15. In this method, the algorithm solves the conflict by using a decision wizard, which selects an element from a list of ambiguous elements. Afterwards all flags are removed and the matching algorithm continues with the execution of the *matchNext*() method, until the list is empty.

Compare elements, that were previously identified to contain an element from the compare state chart, which is also contained in another compare element with a higher similarity, are flagged as *handled* (i.e., in the *checkComparedElement*() method calls). When such an element is identified in Line 8, the algorithm calls the decision wizard in order to solve the conflict (cf. Line 11) and afterwards, tries to match x (cf. Line 12).

4.4. Adapting the Merging Phase

After creating a list of distinct matches in the *Matching Phase*, we have to adapt the *Merging Phase* to merge the identified variability into one model, in order to create a 150% model. The merging is highly dependent on the compared models, because the structure of the model types differs. For example, when merging block-based models, we have to develop an algorithm to merge the blocks, any sub-states, and connectors of the compared models into the final 150% model.

Input: List CE_p with all possible compare elements, $CE_p \subset CE_{p_s}$ **or** $CE_p \subset CE_{p_t}$ **or** $CE_p \subset CE_{p_r}$
Output: List CE_m with all matched compare elements

```
1  method match(CE_p)
2      CE_m ← null;
3      matchNext();
4  return CE_m;
5  method matchNext()
6      if |CE_p| > 0 and !isAmbiguous(next(CE_p)) then
7          x ← next(CE_p);
8          if !isHandled(x) then
9              tryToMatch(x);
10         else
11             processOptionalAndAmbiguousElements();
12             tryToMatch(x);
13         end
14     else
15         processOptionalAndAmbiguousElements();
16     end
17 return;
18 method tryToMatch(x)
19     SBE ← getElementsWithSameBaseElement(x, CE_p);
20     if getComparedElement(x) = ∅ and |SBE| = 0 then
21         match(x);
22         matchNext();
23     else if |SBE| = 0 then
24         checkComparedElement(x);
25         matchNext();
26     else
27         if hasBetterBaseElementMatch(x, CE_p) then
28             matchNext();
29         else
30             if isAmbiguous(x, SBE) then
31                 sort(CE_p);
32                 matchNext();
33             else
34                 checkComparedElement(x);
35                 matchNext();
36             end
37         end
38     end
39 return;
```

Algorithm 4.3.: Pseudocode for the adapted *Matching Phase*

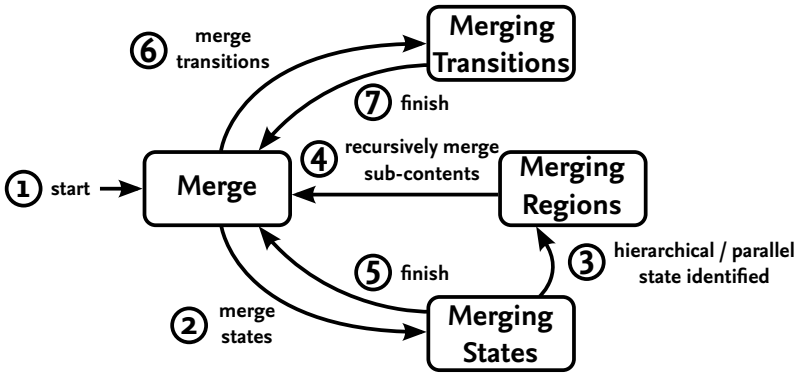


Figure 4.14.: Workflow for the adapted *Merging Phase*

This algorithm is highly adjusted to the structure of the used meta-model. Consequently, we cannot simply adapt the current merge algorithm for block-based models to state charts, but have to develop a suitable algorithm for state charts. Besides, we refactored the workflow for family mining for state charts (cf., Section 4.1) and also compare elements, annotated with variability in previous runs, with new elements. Consequently, the compare elements can contain base elements, which have variability assigned to them (e.g. `(?Transition X,Transition Y)`), which has to be taken into account when merging the compared state charts.

In order to merge the results of compared state charts, we have to find a way, how we can walk through the models and merge all relevant information. For the whole merging process, we need some entity, which checks the calculated similarity of a compare element and defines whether it has to be treated as *mandatory*, *optional*, or *alternative*. Both, the calculation of the similarity and the evaluation of the identified variability are interrelated as they are domain specific for the state charts that should be compared. The only difference is that we calculate the similarity of compare elements during the *Comparing Phase* and have to identify their variability in the *Merging Phase*. Consequently, the entity to evaluate the identified variability should be the metric defined for the compared state charts, because it also defines how the similarity values are calculated.

In order to identify the variability of compared elements, the metric should define thresholds to distinguish, when a compare element is regarded as *mandatory*, *optional*, or *alternative*. Such thresholds could, for example, define that compared elements are regarded as mandatory if their similarity is greater or equal to 95%. Alternative elements could be regarded as such, when their similarity is below 95% but greater than 0%. Consequently, optional elements would be identified as such, when their similarity is exactly 0%. The input for the merging is the list of matched compare elements for the compared models, a copy of the base state chart for the first merging step, or the merged 150% model from a previous iteration if more than two state charts are compared.

In Figure 4.14, we present the workflow to merge the matched results into the final 150% state chart. To start the merging process in 1, we get the root region of the base state chart or the previously merged 150% state chart, the list of matched state compare elements, and the list of matched transition compare elements.

Next, we follow 2 and merge all state compare elements into the root region of the base state chart or the previously merged 150% state chart. During the merging of the states, we check for each of the processed state compare elements, whether it contains hierarchical or parallel states. If none of the merged states is hierarchical or parallel, we follow 5 and 6 to continue with the merging of the

transition compare elements. After finishing the merging of the transition compare elements, we follow 7 and stop the merging algorithm.

If we identify a hierarchical or parallel state during the merging of the state compare elements, we follow 3 and merge the corresponding region compare elements. After merging the regions, we follow 4 and merge all sub-contents for each of the found regions by recursively starting the merge algorithm with the corresponding region, the found list of matched sub-state compare elements, and the found list of matched sub-transition compare elements. After merging the sub-contents of a region, we follow 7 from the last processed step (i.e., the merging of the sub-transition) and continue the state merging process from the previous recursion, where we left to merge the hierarchy.

4.4.1. Merging States

In order to merge state compare elements with existing states, we have to differentiate between multiple cases, since the compare elements can have different variability information assigned to them. Besides, during the merging of the base state chart and the first compared model, no variability is previously set for the base state chart's states, because no previous merging was executed. Consequently, we do not have to consider any information from previous executions. When merging more than two state charts, we run the merging algorithm multiple times and, thus, there is variability set for states contained in the result from the previous merging runs.

We identified the following cases, which have to be covered, in order to merge two models, which have *not* been previously processed to set their variability:

1. The compare element is *mandatory* and compares:
   a) two non-hierarchical and non-parallel states.
   b) two hierarchical states with the same hierarchy.
   c) two parallel states with the same number of regions and the same hierarchy.

   If no variability was previously set, we can process *mandatory* compare elements for non-hierarchical and non-parallel states easily, because we only have to mark the corresponding state in the base state chart as *mandatory*. Compare elements for hierarchical states or parallel states can be handled in the same way. These cases can only occur for states with the same hierarchy and the same number of regions, whose sub-elements are all *mandatory*, because, otherwise, the compare elements would be treated as alternatives, since the compared states would differ to much. Consequently, these hierarchical states and parallel states can also be marked *mandatory*.

2. The compare element is *alternative* and compares:
   a) two non-hierarchical and non-parallel states.
   b) a non-hierarchical and non-parallel state with a hierarchical state.
   c) a non-hierarchical and non-parallel state with a parallel state.
   d) two hierarchical states.
   e) two parallel states.
   f) a hierarchical state with a parallel state.

If a compare element for non-hierarchical and non-parallel states is *alternative*, we have to add the state, which is not already contained in the base state chart, to the corresponding region. Both states have to be marked *alternative* and we have to assign a *group number* to them, in order to identify, which elements are alternative to each other. This group number is needed, in order to distinguish between the different groups, because there can be multiple groups of alternative states in the same region. When processing a compare element, which compares a non-hierarchical and non-parallel state with a hierarchical state or a parallel state, we have to proceed in the same manner as for two non-hierarchical and non-parallel states, because we have to represent the variability of the hierarchy correctly. We could also model the variability in the hierarchy by marking one of the compared states mandatory and adding the sub-contents as optional elements. This solution has advantages, as long as all properties of the compared states are the same, because we would not represent the same information twice (i.e., the states properties).

Problems arise, when the properties differ, because we would loose information when setting the states mandatory, since only one of the states is merged into the base state chart. For example, when we compare a hierarchical `State A` containing property `a` with a non-hierarchical and non-parallel `State B` containing property `b`, we would for example set `State A` mandatory and its sub-contents as optional. Consequently, we would only preserve the property `a`, but loose the information about the existence of property `b`. Thus, this solution is not desirable, unless we develop a way to assign variability options to single properties of states.

When processing a compare element, which compares two hierarchical states or two parallel states the similarity identified by the metric can be ambiguous, because two states can be identified to be *alternative* when the contents of their sub-regions are not fully mandatory to each other, since the overall similarity of these compare elements is calculated by creating the average of the sub-contents' similarities. According to the identified similarity, we would have to mark the compared hierarchical states or parallel states alternative to each other. In this case, we would face problems with mandatory sub-elements, since we would have to add them to both alternatives of the parent state. This is not the desired result and, thus, we regard these "alternative" hierarchical or parallel compare elements as *mandatory* and process their variability by processing their sub-contents correctly.

In Figure 4.15, we present such an example. When comparing the two states `State 1` from both models, we identify, that their name and interface are the same. But as their sub-contents differ (`State 1` in `Model 1` contains `State 2` and `State 1` in `Model 2` contains `State 3`), the calculated overall similarity value for the compare element of the two states `State 1` would not be 100%. Consequently, the two states would be regarded as "alternatives" to each other, although they are the same except for their sub-contents. In order to solve this issue, we regard this compare element as mandatory and process its sub-contents correctly. Consequently, `State 1`, the `Initial` states, and the transitions going from the `Initial` state to `State 1` are marked mandatory. Only the two states `State 2` and `State 3` and the transitions going from state `Initial` to these two states are marked alternative to each other.

The same idea applies, when we compare a hierarchical state with a parallel state, since the variability is contained in the hierarchy of the two compared states. This variability can be

(a) `Model 1` (b) `Model 2`

Figure 4.15.: Example for ambiguous comparisons of hierarchical elements

the different number of regions (i.e., one region versus multiple regions), but also variability in the contents of the regions. Consequently, one of the compared states is marked *mandatory* and the variability in the regions is copied to this state and marked accordingly.

3. The compare element is *optional* and contains the following elements from the base state chart *or* the compare state chart:
   a) a non-hierarchical and non-parallel state.
   b) a hierarchical state.
   c) a parallel state.

   Optional compare elements for non-hierarchical and non-parallel states are also easy to integrate. If the element does not already exist in the base state chart, we have to add the corresponding state to the corresponding region in the base state chart and mark it as *optional* (i.e., the state is only contained in the compare state chart). Otherwise, the state is already contained in the base state chart and only has to be marked as *optional*. The same idea applies for hierarchical states or parallel states, which are added to the corresponding base state chart region, if they were not already contained, and are marked as *optional*. If the state has been copied to the base state chart, all its sub-contents have to be copied to the base state chart as well, and are marked as *mandatory*, since selecting the optional parent state in a product should imply, that all its sub-contents are also selected.

In addition to the previous cases, we identified the following cases for comparisons, where one of the state charts was previously annotated and *already contains variability information*. If the variability was set in a previous merging run, we have to consider the already defined variability during the merging of the next compare elements:

1. The state in the base state chart is *mandatory* and the compare element is *mandatory*. The compare element compares:
   a) two non-hierarchical and non-parallel states.
   b) two hierarchical states with the same hierarchy.
   c) two parallel states with the same number of regions and the same hierarchy.

   This case can only occur, when both states have the same hierarchy and all their sub-contents are also identified to be mandatory, because otherwise the states would be alternative, since they would differ to much. As the base state in the corresponding compare element was previously identified to be *mandatory* and the new compare element is also *mandatory*, we do not have to do anything, since the variability is already marked correctly.

2. The state in the base state chart is *mandatory* and the compare element is *alternative*. The compare element compares:
   a) two non-hierarchical and non-parallel states.
   b) a non-hierarchical and non-parallel state with a hierarchical state.
   c) a non-hierarchical and non-parallel state with a parallel state.
   d) two hierarchical states.
   e) two parallel states.
   f) a hierarchical state with a parallel state.

   If the new compare element is identified to be *alternative* and compares two non-hierarchical and non-parallel states and the previously marked non-hierarchical and non-parallel state is *mandatory*, we have to add the state, which is not already contained in the base state chart. This added state and the previously marked state both are marked *alternative*, and get a group number assigned. This case can occur, when two non-hierarchical and non-parallel states are compared or when a non-hierarchical and non-parallel state is compared with a hierarchical state or a parallel state. When this case occurs for two hierarchical states, two parallel states or a hierarchical state, which is compared with a parallel state, we apply the same ideas as for the case where no variability was previously set. We keep the state, which was previously marked as *mandatory* in the base state chart. If needed, we copy the sub-contents from the compare state chart to this state and set the variability accordingly.

3. The state in the base state chart is *mandatory* and the compare element is *optional*. The compare element contains:
   a) a non-hierarchical and non-parallel state.
   b) a hierarchical state.
   c) a parallel state.

   States that were previously identified to be *mandatory* can become *optional*, when they are not contained in the compared state chart and have to be marked accordingly.

4. The state in the base state chart is *alternative* and the compare element is *mandatory*. The compare element compares:
   a) two non-hierarchical and non-parallel states.
   b) two hierarchical states with the same hierarchy.
   c) two parallel states with the same number of regions and the same hierarchy.

   If the base state in a compare element was previously identified to be *alternative* and the new compare element is identified to be *mandatory*, there is nothing to do, since the variability is already represented correctly, as the alternative state and the compared state are identified to be the same. This can only occur, when the compared states have the same hierarchy, the same number of regions, and all their sub-contents are identified to be mandatory. Otherwise, they would be regarded as alternatives.

5. The state in the base state chart is *alternative* and the compare element is *alternative*. The compare element compares:
   a) two non-hierarchical and non-parallel states.
   b) a non-hierarchical and non-parallel state with a hierarchical state.
   c) a non-hierarchical and non-parallel state with a parallel state.
   d) two hierarchical states.
   e) two parallel states.
   f) a hierarchical state with a parallel state.

   If the base state in a compare element is marked as *alternative* and the new compare element is also identified to be *alternative*, we have to distinguish between different cases. If a non-hierarchical and non-parallel state and a hierarchical state, a non-hierarchical and non-parallel state and a parallel state, or two non-hierarchical and non-parallel states are compared, we apply the same ideas as for a comparison, where no variability was previously set. In this case we can simply copy the new alternative state with all its sub-contents to the base state chart, mark the state alternative, and add the correct group number. If two hierarchical states, two parallel states, or a hierarchical state and a parallel state are compared, we do not change their variability and simply add the missing sub-contents to the corresponding region and mark them and their counter part as alternative. As hierarchical states or parallel states can only be marked as alternative, if they are compared with non-hierarchical and non-parallel states, we would create alternative sub-groups in an alternative state.

6. The state in the base state chart is *alternative* and the compare element is *optional*. The compare element contains:
   a) a non-hierarchical and non-parallel state.
   b) a hierarchical state.
   c) a parallel state.

   When a state was previously identified to be *alternative* and the compare element is identified to be *optional*, the corresponding state from the alternative group was not found in the new compare state chart. Consequently, the whole alternative group has to be marked as *optional*.

7. The state in the base state chart is *optional* and the compare element is *mandatory*. The compare element compares:
   a) two non-hierarchical and non-parallel states.
   b) two hierarchical states with the same hierarchy.
   c) two parallel states with the same number of regions and the same hierarchy.

   If the base state in a compare element was previously identified to be *optional* and the new compare element is identified to be *mandatory*, we do not have to do anything, since the variability is already represented correctly, because the optional state and the compared state are identified to be the same. This can only occur, when the compared states have the same hierarchy, the same number of regions, and all their sub-contents are identified to be mandatory. Otherwise, they would be regarded as alternatives.

8. The state in the base state chart is *optional* and the compare element is *alternative*. The compare element compares:
   a) two non-hierarchical and non-parallel states.
   b) a non-hierarchical and non-parallel state with a hierarchical state.
   c) a non-hierarchical and non-parallel state with a parallel state.
   d) two hierarchical states.
   e) two parallel states.
   f) a hierarchical state with a parallel state.

   If the base state in a compare element is marked as *optional* and the new compare element is *alternative*, we again have to distinguish between different cases. If a non-hierarchical and non-parallel state and a hierarchical state, a non-hierarchical and non-parallel state and a parallel state, or two non-hierarchical and non-parallel states are compared, we can simply copy the new alternative state with all its sub-contents to the base state chart, mark the two states alternative, and add a group number. Besides, the whole alternative group has to be marked *optional*. If two hierarchical states, two parallel states, or a hierarchical state and a parallel state are compared, we only process and copy the sub-contents of the compared states and mark their variability according to the calculations.

9. The compare element is *optional*. The compare elements contains one of the following elements from the compare state chart:
   a) a non-hierarchical and non-parallel state.
   b) a hierarchical state.
   c) a parallel state.

   If a new optional element is identified by a compare element it is part of the newly compared state chart and has to be copied to the base state chart and marked *optional*. In this case the same ideas apply as for a comparison, where no variability was previously set. Consequently, optional hierarchical states or parallel states are copied to the corresponding region and all their sub-contents are marked mandatory.

4.4.2. Merging Regions

Merging regions is closely linked to the merging of states, since states are contained in regions and depend on them. Thus, when merging regions, the identified variability is treated considering the identified variability of the states. We identified the following cases, which have to be covered, in order to merge regions, which have *not* been previously processed:

1. The compared regions are identified to be *mandatory*.

   If the two compared regions are identified to be *mandatory*, they can be directly marked accordingly.

2. The compared regions are identified to be *alternative*.

Two compared regions, which are identified to be *alternative* to each other are also marked *mandatory*, since their variability is defined by their sub-contents and not by the regions themselves.

3. The compared element is identified to be *optional*.

   When a region is identified as *optional* it can either be already part of the base state chart, or be part of the compare state chart. In the first case, we check whether any parent region or parent state is already optional. If so, we mark the region *mandatory*, since it is already contained in an optional or alternative region or state and selecting the corresponding parent should imply selecting the sub-element. Beside, checking for any optional parents, we also have to check for alternative parents, since setting two compared elements alternative creates optional elements, when the two alternative elements are hierarchical. If none of the parents is already marked as optional or alternative, the corresponding region is marked *optional*. In the second case, the element can directly be added to the base state chart and marked *optional*.

In addition to the previous cases, we identified the following cases for comparisons, where one of the state charts was previously annotated and *already contains variability information*:

1. The region in the base state chart is *mandatory* and the compare element is *mandatory*.

   If the base region in a compare element was previously identified to be *mandatory* and the new compare element is also *mandatory*, we do not have to do anything, since the variability is already marked correctly.

2. The region in the base state chart is *mandatory* and the compare element is *optional*.

   Regions that were previously identified to be *mandatory* can become *optional*, when they are not contained in the next compared state chart. In this case, their variability has to be changed from *mandatory* to *optional*.

3. The compared regions are identified to be *alternative*.

   If the compared regions are identified to be alternative, we check whether the base region was previously identified as *optional*, otherwise we set the region *mandatory*, since the alternative variability is contained in the region.

4. The region in the base state chart is *optional* and the compare element is *mandatory*.

   If the base region in a compare element was previously identified to be *optional* and the new compare element is identified to be *mandatory*, we do not have to do anything, since the variability is already represented correctly, because the optional region and the compared region are identified to be the same.

5. The compare element is *optional*.

   If the new compare element is optional, we apply the same ideas as for a comparison, where no variability was previously set and mark the variability accordingly.

4.4.3. Merging Transitions

Merging transitions is highly dependent on the merging of states, since transitions have a source and target state, which has to be marked with the correct variability, before any outgoing or incoming transitions can be merged into the same model. We identified the following cases, which have to be covered, in order to merge transitions, which have *not* been previously processed to set their variability:

1. The compare element is *mandatory* and:
   a) the transition's source state is *mandatory* and its target is *mandatory*.
   b) the transition's source state is *mandatory* and its target is *alternative*.
   c) the transition's source state is *alternative* and its target is *mandatory*.
   d) the transition's source state is *alternative* and its target is *alternative*.

   If the two compared transitions are identified to be *mandatory*, we have to distinguish between different cases. The easiest one is, that the source and the target state both are *mandatory*. Consequently, we have to mark the transition between the states *mandatory*. If one of the source or target states is *mandatory* and the other one is *alternative*, we have to create a copy of the mandatory transition and set the alternative source or target state to reflect the correct variability. For example, if there is a variability for the source state of the transition, we have to create transitions going from the alternative source states to the mandatory target state. The transitions have to be marked *alternative* and we have to assign a group number. If the source and target state both are alternative, we have to create a copy of the mandatory transition for both alternatives (i.e., the combination of the source and target state from the base state chart and for the combination of the source and target state from the compare state chart).

2. The compare element is *alternative* and:
   a) the transition's source state is *mandatory* and its target is *mandatory*.
   b) the transition's source state is *mandatory* and its target is *alternative*.
   c) the transition's source state is *alternative* and its target is *mandatory*.
   d) the transition's source state is *alternative* and its target is *alternative*.

   If the two compared transitions are identified to be *alternative* and the source and target state both are *mandatory*, we have to add the alternative transition, which is not already contained in the base state chart, set the source and target state to the states from the base state chart, and mark both transitions *alternative* and assign a group count to them. The same idea applies, when the source state, the target state, or both states are alternative to each other. In this case, we have to add the alternative transition, which is not already contained in the base state chart, set its source and target state according to the variability, mark the transitions as *alternative*, and set a group count.

3. The compare element is *optional* and:
   a) the transition's source state is *mandatory* and its target is *mandatory*.
   b) the transition's source state is *mandatory* and its target is *optional*.

c) the transition's source state is *optional* and its target is *mandatory*.

d) the transition's source state is *alternative* and its target is *optional*.

e) the transition's source state is *optional* and its target is *alternative*.

f) the transition's source state is *alternative* and its target is *alternative*.

g) the transition's source state is *optional* and its target is *optional*.

For *optional* transitions, we have to distinguish between multiple cases. If the source and target state of an optional transition both are *mandatory* the two states can only be part of a sub-region or sub-state, whose parent was marked optional, since we mark these sub-contents as mandatory. This implies the selection of these mandatory sub-contents, when selecting the optional parent. Consequently, we mark the corresponding transition *mandatory*. If either the source or the target state is marked *mandatory* and the other one is *optional*, we have to mark the transition, linking these states, *optional*. If either the source or the target state is marked *alternative* and the other one is *optional*, we mark the transition *optional*, but do not create another alternative of the transition with the alternative source or target state, since the optionality implies that it has only be added for the variant containing the specific alternative element. The same idea applies, when the source and the target state both are *alternative*. If the source and target state of an optional transition both are *optional*, we can simply add the transition and mark it *optional*.

In addition to the previous cases, we identified the following cases for comparisons, where one of the state charts was previously annotated and *already contains variability information*. These cases are very similar to the identified cases, when no variability was previously set and can be treated mostly in the same manner.

1. The compare element is *mandatory*, and:

   a) the transition's source state is *mandatory* and its target is *mandatory*.

   b) the transition's source state is *mandatory* and its target is *alternative*.

   c) the transition's source state is *alternative* and its target is *mandatory*.

   d) the transition's source state is *alternative* and its target is *alternative*.

   If the source and the target state both are *mandatory*, and the compare element is also *mandatory*, there is nothing to do, since the variability is already represented correctly in the base state chart from a previous comparison. If either the source state, the target state, or both are *alternative*, we have to create an *alternative* between the transitions. In addition, we need to assign the correct group number to these transitions. This might result in alternative groups with more than two transitions, since we might add another transition to an existing group.

2. The compare element is *alternative* and:

   a) the transition's source state is *mandatory* and its target is *alternative*.

   b) the transition's source state is *alternative* and its target is *mandatory*.

   c) the transition's source state is *alternative* and its target is *alternative*.

If the compare element is *alternative*, we have to add either a new alternative to an existing alternative group or create a new alternative group. In both cases, we apply the same ideas for copying the new alternative transition to the base state chart with respect to the variability in the source and target states, as for the cases when no variability was previously identified.

3. The compare element is *optional* and:
   a) the transition's source state is *mandatory* and its target is *optional*.
   b) the transition's source state is *optional* and its target is *mandatory*.
   c) the transition's source state is *alternative* and its target is *optional*.
   d) the transition's source state is *optional* and its target is *alternative*.
   e) the transition's source state is *alternative* and its target is *alternative*.
   f) the transition's source state is *optional* and its target is *optional*.

   If the compare element is *optional*, we check whether the parents of the source and target states are already marked optional or alternative. In these cases, we mark the transition *mandatory*, because we assume that the variability is already represented in the parents of these states. If the previous condition does not apply, we check whether the transition was previously marked as *alternative*. In this case, we additionally mark the transition *optional* to create an optional alternative group. In all other cases, we can simply mark the transition as *optional*.

4.4.4. Pseudocode for the Adapted Merging Phase

In Algorithm 4.4, we present the pseudocode for the merging algorithm. The *merge*() method in Line 1 expects the compare elements for all states and transitions and the base state chart, which should later contain the merged variability. This method starts the merging of the elements by calling the *mergeElements*() method in Line 2 with the given compare elements and the base state chart's root region.

The *mergeElements*() method, first processes all state compare elements ce_s and merges their contents in the given region r (cf. Line 6). Afterwards it checks, whether the currently processed compare element ce_s compares hierarchical or parallel states. In this case, it gets the base state from the compare element ce_s (cf. Line 8) and the corresponding region compare elements and calls the *mergeRegions*() method in Line 9.

The *mergeRegions*() method processes the given region compare elements ce_r one after another and merges the corresponding regions into the given state. First, the method identifies the base region for the corresponding ce_r (cf. Line 19) and gets the sub-contents (i.e., the sub-state and sub-transitions compare elements $CE_{m_{rs}}$ and $CE_{m_{rt}}$) in Line 20 and Line 21. With these elements the *mergeElements*() method is called recursively in Line 22 to merge them into the base state chart. Because of the limited space, we did not show all details of the region merging algorithm, since these algorithms can be implemented using the explanation and enumerations in Subsection 4.4.2.

After processing all states and regions, from possibly hierarchical or parallel states, the algorithm merges all transitions into the corresponding base state chart region (cf. Line 13) by calling the *mergeTransitions*() method. Similar to the merging algorithm of regions, we neglected the details of the *mergeStates*() and *mergeTransitions*() methods because of the limited space.

Input: Lists CE_{m_s} and CE_{m_t} with all matched state and transition compare elements and state chart sc_b
Output: State chart sc_m

```
1  method mergeElements(CE_ms, CE_mt, sc_b)
2  |  mergeElements(rootRegion(sc_b), CE_ms, CE_mt);
3  return sc_m;
4  method mergeElements(r, CE_ms, CE_mt)
5  |  foreach ce_s ∈ CE_ms do
6  |  |  mergeStates(r, ce_s);
7  |  |  if isHierarchicalCE(ce_s) or isParallelCE(ce_s) then
8  |  |  |  s_b ← getBaseElement(ce_s);
9  |  |  |  mergeRegions(s_b, getRegionCEs(ce_s));
10 |  |  end
11 |  end
12 |  foreach ce_t ∈ CE_mt do
13 |  |  mergeTransitions(r, ce_t);
14 |  end
15 return;
16 method mergeRegions(s, CE_mr)
17 |  foreach ce_r ∈ CE_mr do
18 |  |  ...
19 |  |  r_n ← getBaseElement(ce_r);
20 |  |  CE_mrs ← getSubStateCEs(ce_r);
21 |  |  CE_mrt ← getSubTransitionCEs(ce_r);
22 |  |  mergeElements(r_n, CE_mrs, CE_mrt);
23 |  end
24 return;
```

Algorithm 4.4.: Pseudocode for the adapted *Merging Phase*

These algorithms can be implemented using the explanations and enumerations in Subsection 4.4.1 and Subsection 4.4.3.

4.5. Adapting the Family Model Exporter

In this section, we explain the challenges, which arise with the adaption of the current family mining exporter for 150% models of state charts. In order to understand the difficulties, we again have to visualize the structure of block-based models. As explained in Section 2.1 and Section 3.5, the functionality of block-based models is defined by their blocks and their connections do not have a high importance, since they only link the blocks with each other and simply allow the exchange of data without influencing it. Consequently, we do not represent the connections between the blocks. In Section 3.5, we also discussed that state charts are more complex, since not only the states, but also the transitions have an identity, as they influence the functionality of the state charts and have an impact on the results of the calculation. Consequently, we cannot only export the states similar

(a) Hierarchical state (without regions) (b) Hierarchical state (with regions)

Figure 4.16.: Comparison of family model representations for hierarchical states

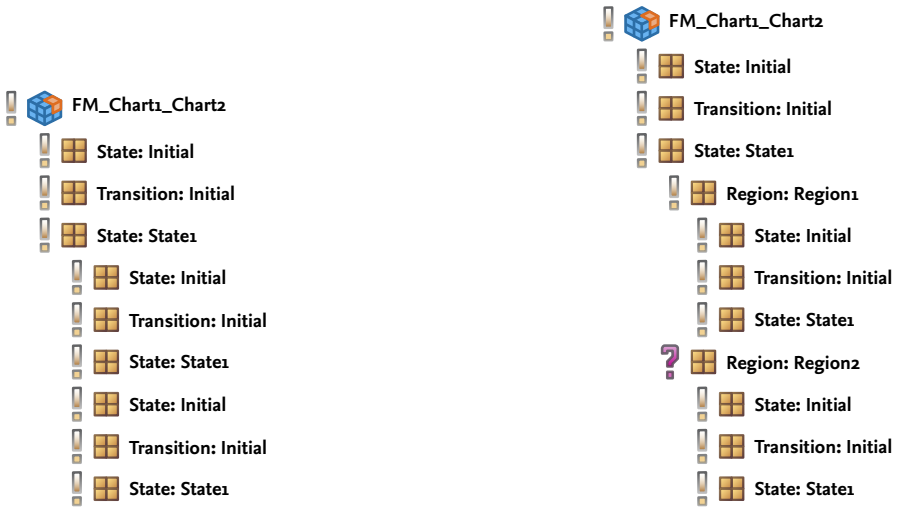
to the blocks from block-based models, but also have to export the transitions, in order to represent all important elements of state charts. Beside states and transitions, we also have to add regions to the family models, because we need a way to group states together, which are part of regions in parallel states. For hierarchical states, it would not be a problem to add the states directly to the sub-hierarchy of the corresponding state. For parallel states, this solution would be ambiguous, because it would not be obvious, which states are part of which region in the parallel state.

We identified these additional elements to be the biggest challenge for the adaption of the family mining exporter. The main reason is, that the family model gets cluttered, because a large number of additional elements need to be visualized. We already faced the same problem, when applying our current approach to industrial scale block-based models with about 1000 blocks and a lot of hierarchical blocks. The large number of elements, which need to be visualized in the family model, make the variability in the compared models hard to understand. Consequently, the advantages for developers, which we try to achieve, are obsolete, since it is nice to identify the variability contained in the related models, but it is not helpful if the results cannot be understood by experts, because the amount of information overwhelms them.

In Figure 4.16, we present two family models for hierarchical states. As we can see in Figure 4.16a, we do not necessarily need regions as additional hierarchy elements in family models to understand, that the sub-states of `State1` are sub-states. In Figure 4.16b, we present the same family model with an additional hierarchy level for `Region1`. This additional hierarchy element has no advantages as long as we only have one region in a state.

In Figure 4.17, we present two family models for parallel states, in order to illustrate the advantages of regions as additional hierarchy elements in family models. As we can see in Figure 4.17a, we cannot understand, which states are part of which region, since they are only sub-states of *State1*. Besides, this representation does not show a legal state chart, because `State1` contains two initial sub-states and two sub-states with the same name, which is not allowed. In Figure 4.17b, we present the same family model with additional region elements. Now, we can understand, which states belong to which region and have a legal state chart representation. Besides, we are now able to mark `Region2` as *optional*.

When using the version without additional region elements, we could also have marked all sub-elements optional, but this would have increased the number of variation points, because each of the elements could be selected or deselected individually, which is not the desired result. Consequently,



(a) Parallel state (without regions)

(b) Parallel state (with regions)

Figure 4.17.: Comparison of family model representations for parallel states

using additional region elements allows us to represent the variability of the compared state charts correctly by marking them accordingly.

As we can see in Figure 4.16 and Figure 4.17, we used labels to illustrate, which elements in the family models are states or transitions. From our point of view, this is only a workaround, since it is not directly clear at a glance, which elements are states or transitions, because their visualization (i.e., the brownish package) in front of them is always the same. Consequently, we argue that a different representation of the elements in state charts should be found.

In Figure 4.18, we present a first idea, how the visualization for family models could be improved. In this family model, we used distinct pictograms for the different elements to better distinguish between them. States are represented by boxes with rounded corners, transitions are represented by an arrow, and regions are represented by a symbol, which is normally used to indicate a line break in texts. These pictograms are only first ideas, how the visualization could be improved, and further research might find more suitable ones.

Another approach to reduce the complexity when analyzing family models for state charts, could be to add transitions as sub-elements of their source states. This allows to more easily understand, where the transitions start, although we still do not directly see the target states. In Figure 4.19, we present an example, how this approach could be realized. We now can directly identify that there are two alternative transitions starting at the `Initial` state and one transition starts at `State1a` and `State1b`, respectively.

When using distinct pictograms for the different elements in family models, we still have the problem that large family models get confusing because of the large amount of information shown. For example, with a lot of differences between the compared models many variation points exist. One approach to overcome these problems could be *variation point enlargement* for the mined state charts, in order to group variable parts with the same variability together. This approach reduces the number of variation points, since, for example, multiple alternative elements are grouped together. In Figure 4.20a, we present a family model, which contains a region, whose sub-contents are all marked alternative (initial states are always regarded as mandatory elements, since they are

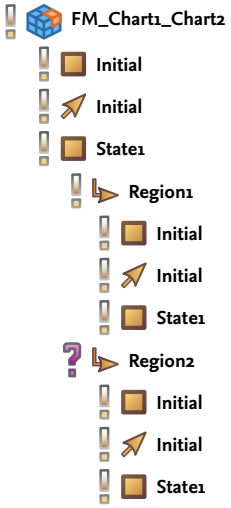


Figure 4.18.: Improved family model using pictograms for the different elements

required in a legal state chart). Consequently, we can apply variation point enlargement and create two regions, which are alternative to each other. These regions contain the corresponding elements, which were previously identified to be alternative to each other. In Figure 4.20b, we show the resulting family model. As we can see, there are two alternative regions `Region 1`, which contain the mandatory initial state `Initial` and the mandatory states `State1` and `State2`, respectively. Besides, they contain the mandatory transition `Initial` from the initial state to the corresponding state.

The created larger variation points allow to directly select one of the two variants (i.e., the variant containing `State1` or `State2`) and, consequently, ease the configuration of product variants. This kind of variation point enlargement should be further investigated, because in some situations it creates undesired restrictions, that could reduce the number of possible variants we can generate. For example, given that the two transitions `Initial` differ in their actions (e.g., action a and action b), which are executed when taking the corresponding transition, we could limit the number of variants when applying variation point enlargement. With the family model in Figure 4.20a it is possible to create four variants with the alternative states and transitions:

1. Transition `Initial` together with action a and `State1`.
2. Transition `Initial` together with action b and `State1`.

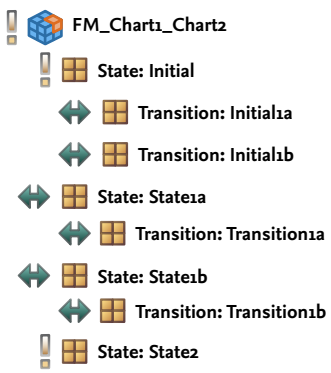
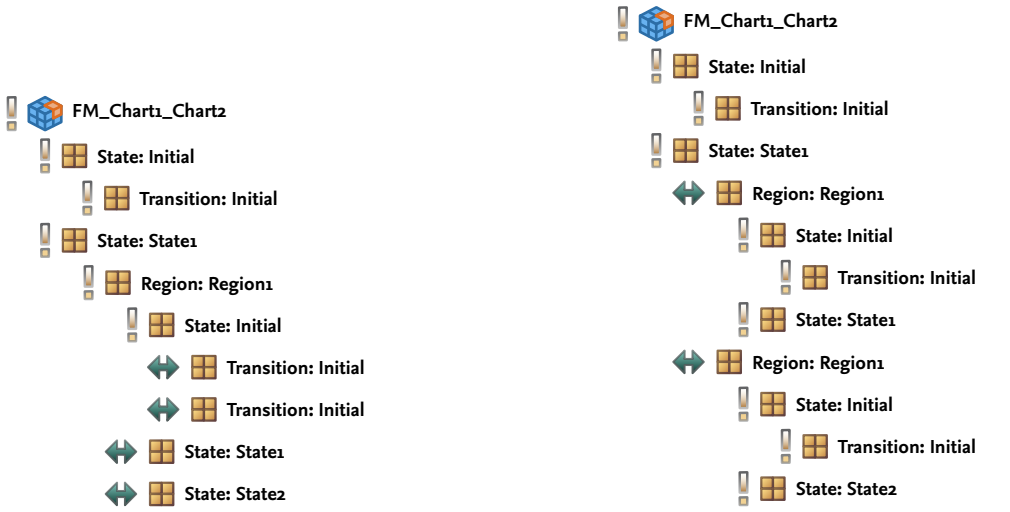


Figure 4.19.: Family model with transitions as sub-elements of their source states



(a) Without variation point enlargement (b) With variation point enlargement

Figure 4.20.: Variation point enlargement

3. Transition `Initial` together with action a and `State2`.
4. Transition `Initial` together with action b and `State2`.

With the applied variation point enlargement in Figure 4.20b, we reduce the number of possible variants, since only two variants are left. Consequently, variation point enlargement should be applied with caution, since it might produce undesired results.

As we can see, the discussed improvements are able to reduce the complexity of the family models and help to easier understand the variability between the compared state charts. However, we still can hardly see, which transitions connect which states. In order to overcome this limitation of the used family model representation, we could use a wholly new approach. For example, we could use the standard state chart visualization and use *annotations* or *colors* to distinguish between the different variants. In Figure 4.21, we present a small example for such a visualization. The shown state chart is a representation for the family model in Figure 4.19. We do not use the standard default transition representation in this state chart, because we need the initial state as an additional state (i.e., `Initial` in this example) to represent the variability in the outgoing transitions. As we can see, all states and transitions, which are mandatory (i.e., part of every product variant) are black and do not have annotations assigned to them. The parts, which are part of a specific product variant are annotated (i.e., with `Chart1` and `Chart2` for this example) and have a color, which is unique for the corresponding product variant (i.e., red for `Chart1` and blue for `Chart2` in this example).

The presented approach could be combined with ideas by Fuhrmann et al. [4], who describe ideas how the visualization of large models can be improved by using sophisticated layout algorithms and a *view management* logic, in order to give developers an efficient overview of the contents. This view management logic collapses all hierarchical elements by default and only expands the sub-contents if the user opens them. This way only a part of the variability is displayed. The user can selectively open the parts, that are currently important for the analysis of the variability. In Figure 4.22a and Figure 4.22b, we present a possible way how the visualization of family models could be improved by the ideas of Fuhrmann et al. [4]. In Figure 4.22a, we show a simple family model for a state chart,

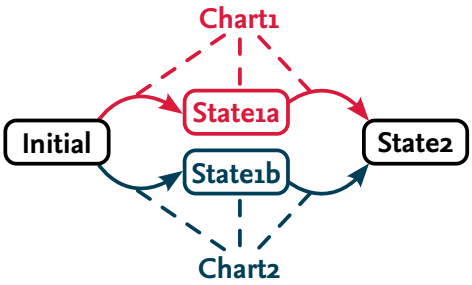
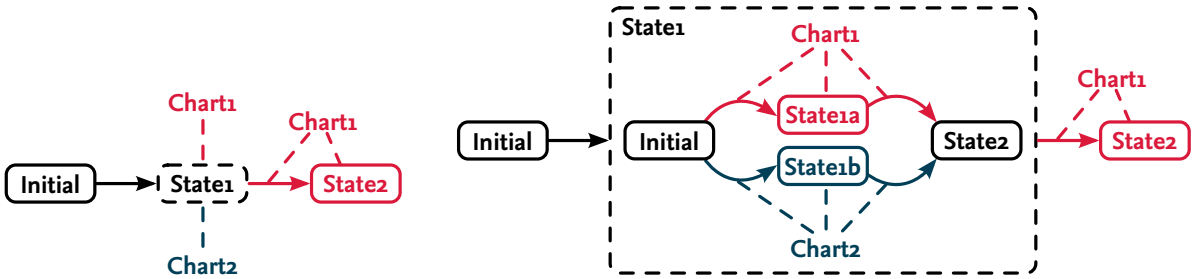


Figure 4.21.: Family model represented as a state chart with annotations and color

which contains a mandatory initial state `Initial`, a mandatory hierarchical state `State1`, and an optional state `State2`. As we can see, the hierarchical state `State1` is collapsed and does not show its sub-contents, but the colored annotations and the dashed border at `State1` indicate, that variability is contained in its hierarchy. When the user, for example, makes a double click on this state, the sub-contents are expanded and the user can analyze the variability in further detail. If the user double clicks again on the corresponding state, the sub-contents are collapsed and only the first hierarchy level of the state chart is displayed. An idea to further reduce the complexity is to hide the contents surrounding the expanded hierarchical state. This way, the user can focus only on the sub-contents. On the other hand, the user might need this information to understand the variability contained in the hierarchical state. Hence, allowing the user to set, whether the surrounding information is displayed could be a good trade-off to allow both solutions depending on the use case. This example also shows a limitation of the representation of family models in standard state charts, because we cannot represent regions, as they are only modeled indirectly in the model. Consequently, marking a region *optional* is only possible with a workaround, such as, adding the question mark indicating the optionality of the region to the contents of the region itself.

The ideas to annotate standard state charts with variability information could be combined with the classical family model view. For example the user could click on an element in the family model representation and the corresponding element in the state chart is highlighted, or the other way round. This way, we can better understand the relation between the state chart and family model elements and might ease the navigation in the representation.

Another approach presented by Meyer [12] introduces layers to display the deltas of a delta-oriented software product line. Deltas are used to modify an existing core product of a software product line



(a) Collapsed view of a hierarchical family model

(b) Expanded view of a hierarchical family model

Figure 4.22.: Hierarchical family model views using some of the ideas by Fuhrmann et al. [4]

by adding, removing, or modifying functionality. These deltas represent the steps that need to be applied in order to create new product variants using the existing core product. The layer approach by Meyer [12] creates a layer for the core product and allows to add layers for each delta, which is introduced to create new product variants. By clicking on the entry for the corresponding delta in the outline of the product line, the user can hide different delta layers to get a better overview of other deltas and to ease the editing process. This approach could be adapted to family models by using the base state chart as a base layer and adding the identified variability between this state chart and others to different layers. This way, the developer can analyze the variability between the base state chart and any other state chart by selecting the corresponding layer and hiding all information, that is not needed for this analysis.

The discussion about possible solutions and about the advantages and disadvantages of the presented approaches to create family models for state charts, shows that this topic is not trivial and needs further thoughts. As this topic is not part of this thesis and finding a suitable solution to represent family models is a complex task, we only present first ideas. In order to find an efficient way to visualize family models for state charts, different ideas have to be compared with each other, and affected user groups should be involved in the process of creating a suitable solution.

4.6. Introducing a new Compare Algorithm for State Charts

The current algorithm described in Section 2.4 and adapted for state charts in the previous sections is highly adjusted to block-based models. During the adaption of this algorithm (in the following referred to as the AdaptedFaMine algorithm) in the previous sections, we reused the basic ideas, which were developed for block-based models, but introduced further algorithms, in order to allow the comparison of different state types (e.g., hierarchical states and parallel states), transitions, and regions. Since the current implementation for block-based models was developed, having only these kind of models in mind, the used data-flow analysis focuses on the characteristics of block-based models. As described in Section 3.5 there are some differences between block-based models and state charts, regarding their identities (i.e., the impact of the corresponding elements on the functionality of the compared models).

In Figure 4.23, we show the workflow for our new algorithm (in the following referred to as the ImprovedFaMine algorithm). The ImprovedFaMine algorithm combines the *Comparing Phase* and the *Matching Phase* into a single phase. The basic idea is to take advantage of some characteristics, which are unique for state charts.

The first difference between block-based models and state charts is the number of start blocks and initial states. While block-based models can have an arbitrary number of start blocks on every hierarchical level, state charts can only have one single initial state per hierarchy. Consequently, we can take advantage of this fact and directly match the initial states of the compared state charts, since this is the only possible combination of these states. As we already described in Section 3.5, the transitions of state charts have an own identity and in contrast to block-based models have to be directly considered with own compare elements in the comparison. Hence, the next step after matching the initial states is to compare the transitions starting at these states. In both versions of the current algorithm, the version for block-based models and the adapted version for state charts (i.e., the AdaptedFaMine algorithm), we first create all compare elements for the whole model, before using the matching algorithm to find distinct matches for the compared models. In our

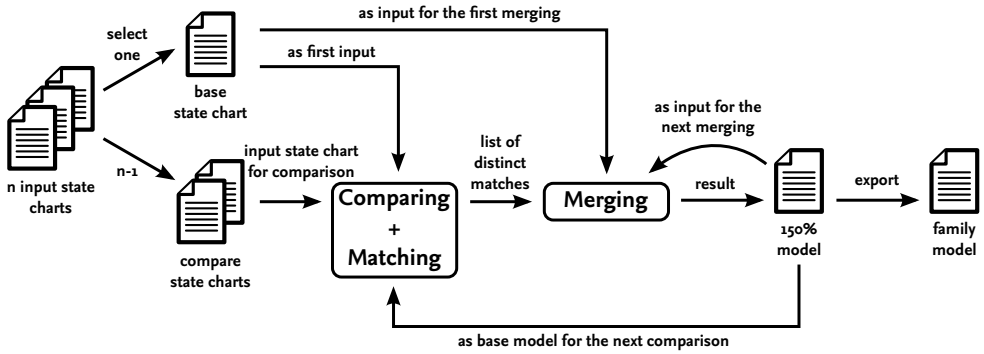


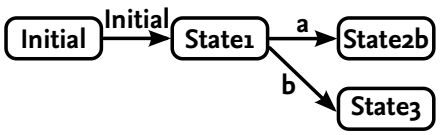
Figure 4.23.: The workflow for the new family mining algorithm

ImprovedFaMine algorithm, we create a list of compare elements for the compared outgoing transitions of the matched initial states. This list of possible compare elements for the outgoing transitions is now matched to find the distinct matches for the transitions. Advantage of this approach is, that we directly know, which subsequent states have to be matched, because each transition always needs to have a source state and a target state, in order to represent a legal state chart transition. Consequently, matching and finding a distinct match for each transition also implies, that we find a distinct match for the subsequent states, because we directly see, which states have to be used to create new compare elements. Since no transition can be matched more than once to another transition, the created compare elements for the subsequent states directly represent distinct matches. For the states used in these distinct compare elements, we again create the list of subsequent transitions and repeat the previously described steps, until the end of the compared state charts is reached and no new compare elements can be created. Similar to the AdaptedFaMine algorithm, we have to check for each iteration, whether the found elements were already considered during the comparison. This approach prevents the algorithm from running into an infinite loop if, for example, a state has a self-loop or a transition back to a state, which was previously processed. For hierarchical states or parallel states, we apply the same comparison ideas as for the AdaptedFaMine algorithm. Consequently, we recursively start the ImprovedFaMine algorithm for new hierarchy levels.

In order to illustrate our ImprovedFaMine algorithm and to compare it with the AdaptedFaMine algorithm, we use the two state charts in Figure 4.24. In this example, we use `State Chart 1` as a base state chart.

In order to compare two models, the first three steps for both approaches are the same. First, we create the compare element `(Initial,Initial)` for the two initial states `Initial` from `State Chart 1` and `State Chart 2`. Second, we create the lists of all subsequent transitions for both initial states, which in this example only contain the transition `Initial` going from state `Initial` to

(a) `State Chart 1`



(b) `State Chart 2`

Figure 4.24.: Example state charts used to illustrate the ImprovedFaMine algorithm

state State1. Third, we create the compare elements for the found transitions (i.e., (Initial,Initial) for this example). At this point the two algorithms start to differ, since the AdaptedFaMine algorithm creates the list of all subsequent states for the found transitions (i.e., (State1,State1) for this example). The ImprovedFaMine algorithm on the other hand, runs the matching algorithm on the created transition compare elements and uses the matched compare elements (i.e., (Initial,Initial) for this example) for the next steps.

While the AdaptedFaMine algorithm, compares each found subsequent state from the base state chart with each found subsequent state from the compare state chart (i.e., (State1,State1) is created), the ImprovedFaMine algorithm directly knows, which subsequent states have to be compared, because the matched transitions' target states (i.e., State 1 for both state charts) can directly be used to create the corresponding compare elements. Next, both algorithms create the list of all subsequent transitions for the created compare elements (i.e., transition a for State Chart 1 and transitions a and b for State Chart 2) and create the corresponding compare elements (i.e., (a,a) and (a,b)). Again, the ImprovedFaMine algorithm executes the match algorithm on these compare elements and identifies (a,a) and (null,b) as the corresponding distinct matches. Consequently, the ImprovedFaMine algorithm directly creates and matches the compare elements (State2a,State2b) and (null,State3), because these are the target states of the previously matched transitions. The AdaptedFaMine algorithm has to identify the lists of subsequent states for the found transitions (i.e., State2a for State Chart 1 and State2b and State3 for State Chart 2) and create the corresponding compare elements (i.e., (State2a,State2b) and (State2a,State3)). As a last step, the AdaptedFaMine algorithm runs the matching algorithm on the resulting lists of state compare elements: (Initial,Initial), (State1,State1), (State2a,State2b), (State2a,State3) and transition compare elements: (Initial,Initial), (a,a), (a,b) and creates the optional elements (null,State3) and (null,b). After this step, both results are the same and the resulting list of distinct matches for the state charts in Figure 4.24 is: (Initial,Initial), (State1,State1), (State2a,State2b), (null,State3) for states, and: (Initial,Initial), (a,a), (null,b) for transitions.

As we can see, the two algorithms create the same list of distinct compare elements for this example. In contrast to the AdaptedFaMine algorithm, our ImprovedFaMine algorithm only runs the comparing and matching algorithm on transitions and uses this information to create the corresponding distinct matches for states. Using this approach, the overall number of compare elements during the matches is reduced, since the algorithm is run on a smaller set of compare elements. On the other hand, the overall number of match algorithm calls is increased, because we call it multiple times instead of only once at the end of the comparison.

In Table 4.1, we compare the ImprovedFaMine algorithm (in the right column) with the AdaptedFaMine algorithm (in the left column) using four characteristics:

- The number of match algorithm calls needed to find the distinct compare elements for states and transition in the compared state charts.
- The overall number compare elements for states and transitions, which are created during the execution of both algorithms.
- The average number of state and transition compare elements, which need to be matched when the matching algorithm is called for the corresponding elements. In order to calculate

	AdaptedFaMine algorithm (states / transitions)	ImprovedFaMine algorithm (states / transitions)
Matching algorithm calls	1 / 1	0 / 2
Number of created compare elements	5 / 4	4 / 4
Average number of compare elements during matching	4 / 3	– / 1.5
Number of distinct compare elements	4 / 3	4 / 3

Table 4.1.: Comparison of the two family mining approaches for the example in Figure 4.24

this characteristic, we count the overall number of compare elements for the compared types (i.e., states or transitions), which are processed during matching. When counting these elements, we leave all compare elements aside, that were created by the matching algorithm after it finished and an element was identified to be optional. For example, during the execution of the ImprovedFaMine algorithm for our example in Figure 4.24, we processed three compare elements during the matching of transitions (`(Initial,Initial)`, `(a,a)`, and `(a,b)`) and created one optional compare element after matching (i.e., `(null,b)`). To calculate the average number of state and transition compare elements, which need to be matched when the matching algorithm is called for the corresponding elements, we divide the counted number of compare elements by the overall number of matching algorithm calls for the corresponding element. Consequently, for our example the average number of transition compare elements, which need to be matched when the matching algorithm is called is 1.5, since we called the matching algorithm twice for transition compare elements and counted three transition compare elements.

- The number of distinct compare elements for states and transitions (i.e., the number of compare elements, which are kept after matching).

As we can see, the AdaptedFaMine algorithm only calls the matching algorithm twice, once for the compared states and once for the compared transitions. Our ImprovedFaMine algorithm only calls the matching algorithm to match transitions, because states are matched implicitly, but needs to call the matching algorithm more often than the AdaptedFaMine algorithm, since it processes the compare elements in *segments*. After comparing the initial states of two state charts, a new segment is created, which compares and matches the subsequent transitions. Afterwards, we can directly compare and match the target states of the compared transitions and create a new segment for the subsequent transitions. These steps are repeated until the state charts are completely processed. As we can see, the match algorithm needs to be called for each segment, in order to match the transitions. For our example, the match algorithm is only called twice. Of course, this number increases when larger state charts are compared, since more segments are needed to process the state charts.

Advantage of our ImprovedFaMine algorithm is that we only use the matching algorithm for transitions and, consequently, can only have *ambiguous elements* for these transitions, since states are directly matched according to the matched transitions. Ambiguous elements occur, when compare elements exist, that have the same similarity and contain the same base element or compare element. In these cases, we cannot directly match one of the elements, but have to mark the corresponding elements *ambiguous* and hope, that other matched elements help to solve these conflicts.

If a conflict cannot be solved automatically, the corresponding elements are presented to the user and a manual decision has to be found. As we only apply the matching algorithm for small segments of the model, we further reduce the chance of ambiguous elements, because every time we call the matching algorithm, only a small subset of all possible compare elements needs to be matched. For our example, we have to match four state compare elements and three transition compare elements during the *Matching Phase* of the AdaptedFaMine algorithm. The average number of compare elements during the matching phases for our ImprovedFaMine algorithm is lower, since we do not have to match states, because they are compared and matched implicitly, when we use the target states of the matched transitions. During the matching of the compared transitions, we only have to match a subset of the overall transition compare elements. Consequently, the average number of transition compare elements, which are matched, is 1.5 for our example.

Another big advantage compared to the AdaptedFaMine algorithm is the number of created compare elements, since the ImprovedFaMine algorithm needs to create less compare elements for compared states. In Table 4.1, we can see, that the AdaptedFaMine algorithm creates five compare elements to find distinct matches for states. That is because, we have to create all possible combinations for the segments of states, `(Initial,Initial)` for the first segment, `(State1,State1)` for the second segment, and `(State2a,State2b)` and `(State2a,State3)` for the last segment. After running the matching algorithm on these four compare elements `(State2a,State3)` is ruled out and we have to create the compare element `(null,State3)` to represent the optional state.

Our ImprovedFaMine algorithm, can use the results of the *Matching Phases* for the transition segments, because the matched transitions allow us to directly create the correct compare elements for their target states. Consequently, we can directly create the compare element `(null,State3)` for our example and do not need to create the incorrect compare element `(State2a,State3)`. The number of created transition compare elements is the same for both approaches, because we still need to execute the *Comparing Phase* and the *Matching Phase* for transitions in both approaches. In our example, we reduced the number of created state compare elements by one element, but with larger state charts the effect should be higher, especially when a large number of combinations needs to be created for a segment.

The presented numbers only give a small insight into the differences of the algorithms and possible advantages of the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm. The algorithms should be further evaluated and compared with larger models, in order to better understand the differences between the approaches.

In order to realize the workflow for the ImprovedFaMine algorithm in Figure 4.23, we have to interleave the execution of the `Comparing Phase` and the `Matching Phase`, because we directly call the matching algorithm on the created transition compare elements of the processed segments. The *Matching Phase* can remain unchanged compared to the *Matching Phase* of the AdaptedFaMine algorithm, because we want to use the same ideas for the matching algorithm, but call it on a smaller set of compare elements. The *Merging Phase* also remains unchanged compared to the NewWorkflow of the AdaptedFaMine algorithm (cf. Figure 4.3), because the structure of the generated results is the same and we can use the same algorithms to merge them back into one model. Consequently, we can also reuse an implementation of a possible exporter for a family model representation. As we can see, only the algorithms for the *Comparing Phase* change for the new family mining algorithm, which eases the implementation, because most parts can be reused.

4.6.1. Pseudocode for the Improved Algorithm

In Algorithm 4.5, we present the pseudocode for the ImprovedFaMine algorithm. This algorithm combines the *Comparing Phase* and *Matching Phase*. It is started by calling the *compare*() method in Line 1 with two state charts sc_b and sc_c as input, which represent the base state chart and the compare state chart, respectively. This method initializes the lists CE_{m_s} and CE_{m_t} in Line 2 and Line 3, which store the matched state and transition compare elements. Afterwards the initial states from sc_b and sc_c are compared and directly matched by calling the *compareStates*() method in Line 4.

The *compareStates*() method checks, whether one of the compared states is *null*. In Line 8, two states are compared, which are not *null* and in Line 10 and Line 12 a state from the base state chart, which is not *null* is compared with a state from the compare state chart, which is *null*, or the other way round. In contrast to the algorithm in Algorithm 4.1, the created compare elements are not stored in a list of possible compare elements, but are stored in the list of matched state compare elements CE_{m_s}, since they represent distinct matches (cf. Section 4.6). After creating the corresponding compare elements, the algorithm creates in Line 14 and Line 15 the lists of successor transitions for the compared states and passes them in Line 16 to the *compareTransitions*() method.

This method is very similar to the implementation in Algorithm 4.1 except for the integrated matching part. The algorithm first checks, whether the two lists $succ_{b_t}$ and $succ_{c_t}$ passed to this method are empty. If both lists contain at least one transition (cf. Line 20), all possible combinations of these states are created in Line 23. If one of the lists is empty, (cf. Line 26 or Line 30) the elements from the other non-empty list are compared with *null* (cf. Line 28 and Line 32). All created compare elements are stored in the list of possible compare elements for states (CE_{p_t}).

In Line 35, this list of all possible transition compare elements is matched by calling the *match*() method from Algorithm 4.3 and storing the partial list of matched transition compare elements in PCE_{m_t}. In Line 36, this list is added to the list CE_{m_t} of overall matched transition compare elements. Each of the transition compare elements from PCE_{m_t} is processed in Line 38 to create the corresponding state compare elements.

As the matching algorithm and the merging algorithm are the same for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm, we do not explain these algorithms again, but reference Subsection 4.3.1 and Subsection 4.4.4 for the corresponding explanations.

4.7. Summary

In this chapter, we discussed an identified problem with the current approach, which can produce inaccurate results in certain situations. As a result, we decided to refactor the workflow and to execute all three phases (i.e., the *Comparing Phase*, the *Matching Phase*, and the *Merging Phase*) for two compared models and use the merged results as an input for the comparison with the next model and the next merging. This differs from the current implementation, as we currently compare all compare models with the defined base model and match the results of each comparison. The matched results are then used to create a merged 150% model, which might be inaccurate, because the variability is only identified between the base model and each of the compare models. Next, we explained, how the current family mining approach for block-based models can be adapted for state charts (we called this adapted algorithm the AdaptedFaMine algorithm) and which challenges have to be tackled during the adaption of the different phases. For example, new compare elements

Input: Two state charts $sc_b, sc_c \in SC, sc_b \neq sc_c$
Output: Lists CE_{m_s} and CE_{m_t} with all matched compare elements for states and transitions

```
1  procedure compare(sc_b, sc_c)
2      CE_ms ← ∅;
3      CE_mt ← ∅;
4      compareStates(initial(sc_b), initial(sc_c));
5  return;

6  procedure compareStates(s_b, s_c)
7      if s_b! = null and s_c! = null then
8          CE_ms ← compare(s_b, s_c);
9      else if s_c = null then
10         CE_ms ← compare(s_b, null);
11     else if s_c = null then
12         CE_ms ← compare(null, s_c);
13     end
14     succ_bt ← succ(s_b);
15     succ_ct ← succ(s_c);
16     compareTransitions(succ_bt, succ_ct);
17 return;

18 procedure compareTransitions(succ_bt, succ_ct)
19     CE_pt ← ∅;
20     if |succ_bt| > 0 and |succ_ct| > 0 then
21         foreach t_b ∈ succ_bt do
22             foreach t_c ∈ succ_ct do
23                 CE_pt ← compare(t_b, t_c);
24             end
25         end
26     else if |succ_bt| > 0 then
27         foreach t_b ∈ succ_bt do
28             CE_pt ← compare(t_b, null);
29         end
30     else if |succ_ct| > 0 then
31         foreach t_c ∈ succ_ct do
32             CE_pt ← compare(null, t_c);
33         end
34     end
35     PCE_mt ← match(CE_pt);
36     CE_mt ← PCE_mt;
37     foreach ce_t ∈ PCE_mt do
38         compareStates(getBaseTarget(ce_t), getCompareTarget(ce_t));
39     end
40 return;
```

Algorithm 4.5.: Pseudocode for the improved *Comparing Phase* and *Matching Phase*

have to be introduced for regions and transitions, which then have to be created by the compare algorithm. Furthermore, the matching algorithm has to be modified to support the matching of state compare elements, region compare elements, and transition compare elements. In order to compare states with each other, we described algorithms to compare hierarchical states and parallel states with other states. We created a new merging algorithm to generate correct results, because block-based models and state charts differ to much (e.g., for state charts we also have to represent the correct variability of the transitions). As a last step for the adaption, we discussed the limitations of the current family model representation and proposed some ideas, how a better visualization can be found. However, we do not present a final solution, as this thesis only focuses on the adaption of the basic algorithms to apply family mining to state charts. Beside describing how the existing algorithm can be adapted, we introduced the ImprovedFaMine algorithm to compare state charts with each other. This algorithm combines the *Comparing Phase* and the *Matching Phase* and takes advantage of the structure of state charts, which might reduce the number of compare elements created during the comparisons. The algorithm might also reduce the number of elements, which have to be matched during the *Matching Phase*, which in turn might reduce the number ambiguous elements.

5 Implementation

In this chapter, we describe the current implementation of the family mining for state charts, as well as the implementation of a tool for the import of *IBM Rational Rhapsody* state charts into our internal meta-model representation. This tool is essential for our evaluation in Chapter 6, since the state charts from the case study, which we want to use during this evaluation, were created with *IBM Rational Rhapsody*.

In Section 5.1, we explain the basic structure of the environment, which we used to implement the ideas of Chapter 4. As we implemented the algorithms using Eclipse plugins, we describe the dependencies between them and their basic contents. In Section 5.2, we explain the functionality of the `gui` plugin, which provides means to start the implemented algorithms for selected state charts. In Section 5.3, we explain the `model` plugin, which provides the meta-model introduced in Section 3.4. In Section 5.4, we explain the `common` plugin, which we realized to provide common classes (e.g., exceptions) that are used by multiple other plugins. In Section 5.5, we explain the `statistic` plugin, which provides classes that are used to calculate statistics for the algorithms, in order to compare the AdaptedFaMine algorithm with the ImprovedFaMine algorithm. In Section 5.6, we explain how we realized the algorithms discussed in Chapter 4. And finally, in Section 5.7, we explain the details of the implementation for the tool, which allows to import *IBM Rational Rhapsody* state charts and store them in our internal meta-model representation.

5.1. Environment of the Implementation

The family mining implementation for state charts is realized using the Java Development Kit (JDK) in version 8[1] by creating Eclipse Rich Client Platform (RCP) plugins for Eclipse Luna[2]. Eclipse is an *integrated development environment (IDE)* for different programming languages, which is extensible with these RCP plugins. Such a plugin consists of standard Java code and contains settings files, which allow to define dependencies to other plugins, to export packages from the corresponding plugin during runtime, and to extend *extension points* defined by Eclipse or other plugins. Exporting packages during runtime allows to access code of other plugins. Defined extension points are used to add functionality to the existing platform. For example, a plugin can extend an extension point to add new buttons to the *graphical user interface (GUI)* of the IDE, which can trigger the execution of the developed code. Before starting a plugin, Eclipse checks whether the defined dependencies are satisfied and, otherwise, prevents the execution. The RCP allows to create modular architectures with plugins depending on each other and, thus, facilitates the efficient reuse of code, because of the created modularity.

In Figure 5.1, we present the architecture of the family mining implementation for state charts with all dependencies between the different plugins. A directed transition between two plugins indicates, that the plugin, in which direction the arrow is pointing, is a dependency of the other

[1] https://www.oracle.com/java/
[2] https://www.eclipse.org/

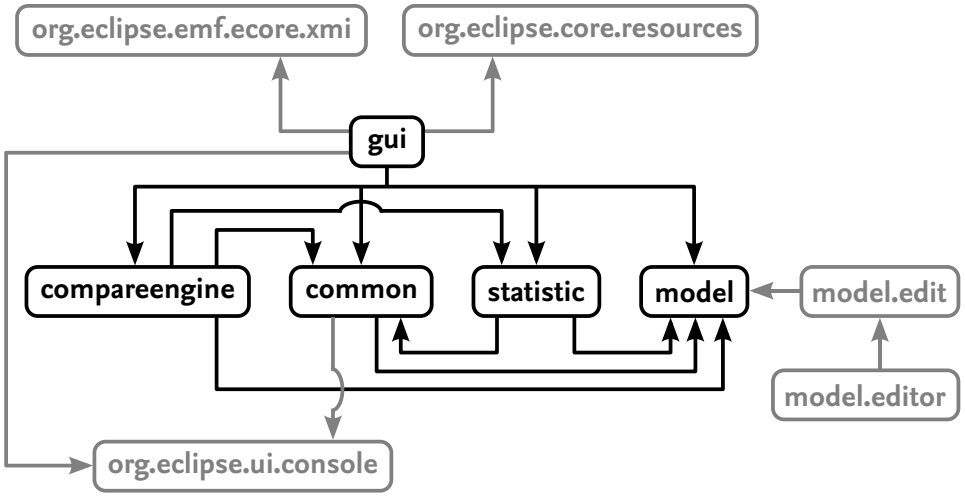


Figure 5.1.: Dependencies of the created Eclipse plugins for the family mining implementation

plugin (e.g., the `compareengine` depends on the `model`). As we can see, the architecture in Figure 5.1 consists of five plugins, whose full name normally has `de.tu_bs.cs.isf.statecharts` as a prefix, which we omitted to increase the readability of the architecture. The architecture also has three additional dependencies to plugins, which are provided by Eclipse.

The basic purpose of the plugins shown in Figure 5.1 is described in the following overview, and will be further explained in the following sections.

gui

This plugin extends the `org.eclipse.ui.commands`, `org.eclipse.ui.handlers`, and `org.eclipse.ui.menus` extension points and creates a new menu in the menu bar, which contains two buttons to trigger the family mining approach by executing the AdaptedFaMine algorithm or the ImprovedFaMine algorithm. The details of this plugin are further explained in Section 5.2.

model

This plugin contains the *meta-model* described in Section 3.4, which was created using the Eclipse Modeling Framework (EMF)[3]. EMF allows to create meta-models with a special editor, which can be used to define classes with attributes and references between these classes. The created meta-model then is used to generate Java classes, which can be used to create model instances of parsed state charts and is the expected input format for our family mining implementation. The details of this plugin are further explained in Section 5.3.

model.edit / model.editor

These plugins are not directly needed for the family mining implementation, but provide methods and classes to display and edit XMI files in the `*.statechart` format.

common

This plugin contains classes, which are used in different other plugins. These contain exceptions created to give detailed information about occurring errors, classes to set the variability

[3] http://www.eclipse.org/modeling/emf/

for states, regions, or transitions of the meta-model, and a console implementation, which can be used for debug purposes. The details of this plugin are further explained in Section 5.4.

statistic
This plugin contains classes, which can be used to create statistics when running the family mining algorithms. The details of this plugin are further explained in Section 5.5.

compareengine
This plugin contains the implementation of the AdaptedFaMine algorithm and the ImprovedFaMine algorithm. The details of this plugin are further explained in Section 5.6.

org.eclipse.core.resources
This plugin is used to access certain resource handling classes of Eclipse, which allow to access the files selected in the *Package Explorer* or *Project Explorer* of Eclipse.

org.eclipse.emf.ecore.xmi
This plugin is used to read *XML Metadata Interchange (XMI)* files, a special *Extensible Markup Language (XML)* format, which can be used to store models created with the defined meta-model.

org.eclipse.ui.console
This plugin is used to realize the debug console provided by the `common` plugin.

5.2. Plugin: gui

The `gui` plugin provides the classes, which start the two algorithms for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm. Therefore, it extends the `org.eclipse.ui.commands`, the `org.eclipse.ui.handlers`, and the `org.eclipse.ui.menus` extension points to create a new menu bar entry, which is called "Family Mining". The created menu contains two buttons "Run State Chart Mining (Adapted)" and "Run State Chart Mining (Improved)", which trigger the corresponding code to run the family mining on state charts for the AdaptedFaMine algorithm (i.e., "Adapted") and the ImprovedFaMine algorithm (i.e., "Improved"). For both buttons, we have created a so called *handler class*, which starts the execution of the code. In order to register this new handler class as a new menu entry, we have to edit the `plugin.xml` file provided by the `gui` plugin. In Listing 5.1, we show an excerpt of the current `plugin.xml` file to explain how new entries can be created by using the example of the "Run State Chart Mining (Adapted)" menu entry.

Any new command for the menu has to be realized in the same way as this entry. Similar to the command in Line 10 a new command tag has to be added, declaring a *unique command id* (cf., Line 11), which identifies the new command. In addition a name has to be entered for the new menu entry (cf., Line 12). Next, a new handler has to be registered by adding a new handler tag (cf., Line 18). This handler reacts to the previously defined command id. The handler defined in Line 18 links the corresponding handler class (cf., Line 19) with the previously defined unique command id (cf., Line 20). For the linked class, we have to provide the full package path (e.g., `de.tu_bs.cs.isf.statecharts.gui.handlers.RunHandlerAdapted` in our example) in order to enable the extension point to find the class. The last step is to add the newly defined command to the created menu. In Line 32, we add the previously defined command to the menu by adding a new

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.4"?>
3  <plugin>
4     <extension
5           point="org.eclipse.ui.commands">
6        <category
7              id="de.tu_bs.cs.isf.commands.category"
8              name="Family Mining">
9        </category>
10       <command
11             id="de.tu_bs.cs.isf.gui.commands.runCommandAdapted"
12             name="Run State Chart Mining (Adapted)">
13       </command>
14        ...
15    </extension>
16    <extension
17          point="org.eclipse.ui.handlers">
18       <handler
19             class="de.tu_bs.cs.isf.statecharts.gui.handlers.RunHandlerAdapted"
20             commandId="de.tu_bs.cs.isf.gui.commands.runCommandAdapted">
21       </handler>
22        ...
23    </extension>
24    <extension
25          point="org.eclipse.ui.menus">
26       <menuContribution
27             allPopups="false"
28             locationURI="menu:org.eclipse.ui.main.menu?after=additions">
29          <menu
30                id="de.tu_bs.cs.isf.gui.menus.familyMenu"
31                label="Family Mining">
32             <command
33                   commandId="de.tu_bs.cs.isf.gui.commands.runCommandAdapted"
34                   id="de.tu_bs.cs.isf.gui.menus.runCommandAdapted"
35                   style="push">
36             </command>
37              ...
38          </menu>
39       </menuContribution>
40    </extension>
41 </plugin>
```

Listing 5.1.: Adding a new run command to the `plugin.xml`

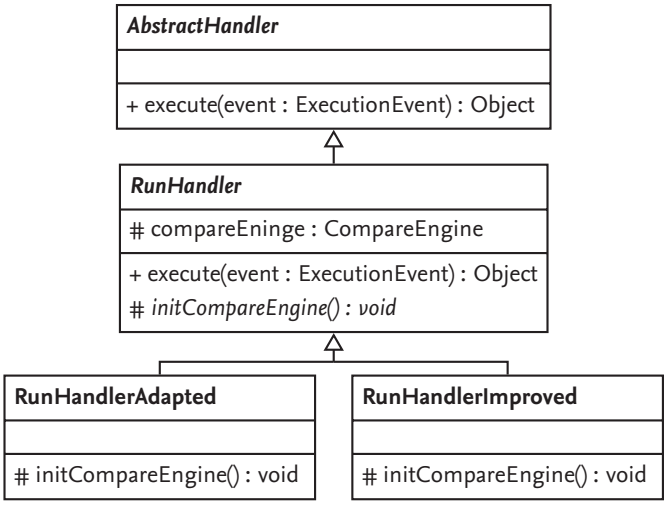


Figure 5.2.: Class diagram of the `gui` plugin

command tag inside the menu tag. The command contains the unique command id (cf., Line 33), which we previously defined, and a new *unique id*, which identifies the command inside the menu (cf., Line 34). Besides, we have to declare the style of the new entry (cf., Line 35). Using the `push` style, we declare a normal menu entry, which is activated by a simple click.

Since the way to start the execution of both approaches is fairly similar, we created the inheritance hierarchy in Figure 5.2. At the top of the hierarchy we have the `AbstractHandler` class, which is provided by Eclipse and allows to start the execution of the two approaches by overriding the `execute()` method in the `RunHandler` class. The `RunHandler` class executes all code, that is common for the start of the two approaches. This includes receiving the list of files containing the state charts, which should be compared, parsing these files, and setting up the statistic. The only method, which needs to be implemented, when extending the `RunHandler` class, is the `initCompareEngine()` method. This method initializes the `compareEngine` variable, which contains the compare engine, that should be used to compare the selected state charts with each other.

By selecting one or more `*.statechart` files in the *Package Explorer* or *Project Explorer* of Eclipse and selecting an approach from the "Family Mining" menu, we receive the list of files, that should be compared. These `*.statechart` files are a special XMI format to store state chart instances, which were modeled using our meta-model described in Section 3.4.

In Listing 5.2, we show some example code, how an XMI file in the `*.statechart` format can be parsed. In Line 9 the file is loaded, by passing a `FileInputStream` and an empty `HashMap` to the `XMIResourceImpl`. The input stream should contain the file, that we want to load and the map is used to configure the parsing of the file. Since we are using the default settings, we do not have to add any parameters to this map. In Line 19, we access the state chart from the parsed file. As each of the files contains exactly one state chart, we can directly access the first resource (i.e., by using `getContents().get(0)`).

After parsing the models, the comparison of the models is started according to the selected approach. All details of the implementation of the *Comparing Phase*, *Matching Phase*, and *Merging Phase* for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm are described in Section 5.6. At the end of the execution, the statistics for the executed approach are displayed.

```
1  // the file that should be parsed
2  File file = new File("/path/to/file.statechart");
3
4  // create a new resource
5  XMIResourceImpl resource = new XMIResourceImpl();
6  try {
7    // read the contents from the file into the resource
8    FileInputStream stream = new FileInputStream(file)
9    resource.load(stream, new HashMap<Object, Object>());
10
11   // close the stream
12   stream.close();
13 }
14 catch (FileNotFoundException | IOException e) {
15   e.printStackTrace();
16 }
17
18 // fetch the state chart from the resource
19 StateChart chart = (StateChart) resource.getContents().get(0);
```

Listing 5.2.: Parsing an XMI file in the `*.statechart` format

5.3. Plugin: model

The `model` plugin provides the classes for the meta-model, which we described in Section 3.4. These classes are generated using EMF, which processes the `statecharts.genmodel` file in the plugin's `model` folder, in order to create the defined classes. The `statecharts.genmodel` file allows to configure the generation process and uses the `statecharts.ecore` definition file, which contains the information about the EMF meta-model that should be generated (cf., Section 3.4).

The generated classes are used in different steps of the family mining process. First, the models, which should be compared with each other, need to be created in the EMF meta-model format, in order to be compatible with our implementation. Consequently, models have to be parsed and transformed into a meta-model representation. One way is to parse and process the models and directly use the created objects for a comparison. Another possibility is to parse the models, create the meta-model representation, and store these results in an XMI file in the `*.statechart` format.

In order to understand, how objects can be created using the generated classes, we show the generation of a new `StateChart` object in Listing 5.3. Objects of the generated classes are not created using the `new` operator, but by calling a factory, which provides methods to create objects for all classes defined by the meta-model. In Line 2, we first fetch the factory object and then create, for example, a `StateChart` object in Line 5.

After using the factory to create a representation of some parsed state charts, we can use the code in Listing 5.4 to store the meta-model representation in an XMI file in the `*.statechart` format. In Line 6, we add the meta-model representation of the state chart, that should be stored, to a new resource (i.e., by using `getContents().add(chart)`). In Line 11, this resource is stored in a file by passing a `FileOutputStream` and an empty `HashMap` to the `XMIResourceImpl`. Similar to loading an XMI file, the output stream contains the file, that should contain the contents of the exported

```
1 // fetch the factory instance
2 StateChartsFactory factory = StateChartsFactory.eINSTANCE;
3
4 // create a new StateChart
5 StateChart chart = factory.createStateChart();
```

Listing 5.3.: Creating a new `StateChart` object

```
1  // the state chart that should be stored
2  StateChart chart =  ...
3
4  // add the contents to a new resource
5  XMIResourceImpl resource = new XMIResourceImpl();
6  resource.getContents().add(chart);
7
8  try {
9    // store the resource's contents into a file
10   FileOutputStream stream = new FileOutputStream("example.statechart");
11   resource.save(stream, new HashMap<Object, Object>());
12
13   // close the stream
14   stream.close();
15 }
16 catch (FileNotFoundException | IOException e) {
17   e.printStackTrace();
18 }
```

Listing 5.4.: Storing a meta-model representation as an XMI file in the `*.statechart` format

resource, and the empty hash map defines that we want to use the default configuration for storing the state chart. The newly created file can be parsed as described in Section 5.2.

The `statecharts.genmodel` file also allows to generate two additional plugins. The `model.edit` plugin provides classes, which are used by the `model.editor` plugin to open, display, and modify `*.statechart` files. These two plugins are not directly needed for the family mining implementation, but are very useful to analyze models, which are stored in XMI files in the `*.statechart` format.

5.4. Plugin: common

The `common` plugin provides different classes, which are used by multiple other plugins. By creating this separate plugin, we increased the modularity and the reuse of code in plugins. The plugin contains three functionalities used by other classes:

1. A set of exceptions, which are used to provide proper error handling.
2. Two static classes, which provide methods to set the variability and group of states, regions, and transitions.
3. A debug console, which allows to show debug messages during family mining.

5.4.1. Exceptions

The `exceptions` package contains a set of exceptions, which are used at different points of the family mining approach to indicate that some error occurred and to give a proper error message for this error. The following exceptions can occur:

BothComparedElementsNullException
: This exception indicates the detection of two elements, which are `null`, and which are compared during the creation of a compare element.

CalculationAlreadyExistsException
: This exception indicates, that during the initialization of a new calculation a value is reinitialized. By using this exception, we prevent overwriting a value, which was previously set.

IllegalCompareException
: This exception indicates, that two elements are compared, which are not comparable to each other. For example, a state is compared with a transition.

IllegalMetricException
: This exception indicates, that a metric was identified to be illegal. For example, some mandatory attribute in the metric is missing or the total value of the metric is greater than 1.0, which is not legal as the values in the interval of 0 and 1.

MergeFailedException
: This exception indicates, that the merging process failed. For example, during the merging of an optional transition the corresponding source or target state is not found, because something went wrong, when merging them into the base model.

MultipleInitialStatesException
: This exception indicates, that a region contains more than one initial state.

NoInitialStateSetException
: This exception indicates, that a region does not contain an initial state.

NoModelsAddedException
: This exception indicates, that no models were defined for the family mining approach.

NonExistingCalculationAccessException
: This exception indicates, that during the calculation of the similarity for two compared elements a value was accessed, which does not exist.

NotInitializedException
: This exception indicates, that some essential element for the family mining approach was not initialized. For example, compare element factories, which are used to create new compare elements, need to have access to the metric, that should be used. If this access is not granted, this exception occurs.

NotMatchedException
: This exception indicates, that the matched results are accessed before they are actually matched.

5.4.2. Variability Options and Groups

The `VariabilityOptions` class contains static methods, which are used to set the variability (i.e., mandatory, optional, or alternative) for state chart elements (i.e., states, regions, and transitions), and to check what kind of variability was assigned. Besides, a helper method exists to print the variability of an element to the console. We use the following symbols, which are based on the family model representation of *pure::variants* to print the variability of elements to the console:

- exclamation mark (i.e., `!`) for mandatory elements
- question mark (i.e., ?) for optional elements
- double arrow (i.e., <=>) for alternative elements

The `VariabilityOptions` class also provides methods to create a statistic for state charts, which counts the number of mandatory, optional, and alternative states, regions, and transitions.

The `GroupOptions` class provides static methods to define group numbers for alternative elements, and to check whether the group number was set and what value it has. Besides, alternative groups can be set optional.

Both classes, the `VariabilityOptions` class and the `GroupOptions` class, use the map defined by the `ParameterizedElement` class and use a number of `Strings` as keys to store additional parameters. In Table A.1 in Section A.1, we show the list of all these keys. This list should be checked before introducing new keys, in order to prevent conflicts and overriding parameters.

5.4.3. Console

During the development of the code, the plugins are not *deployed* (i.e., added to an Eclipse installation) as `*.jar` files, but are executed in an Eclipse runtime, which is started from the normal Eclipse instance. The `Console` class contains static methods to print `Strings` to the console of the Eclipse runtime instance, because normal `System.out.println(...)` commands print to the console of the Eclipse instance from which the Eclipse runtime instance was started. The `Console` class can easily be used by adding the `common` plugin as a dependency to another plugin and calling `Console.println(...)`. The `Strings` passed to this method are directly printed to the console of the Eclipse runtime instance. This behavior can be disabled and enabled by using the `setEnabledDebug(...)` method. When printing to the console of the Eclipse runtime instance is disabled, the method has the same behavior as `System.out.println(...)` and prints to the console of the Eclipse instance, which started the Eclipse runtime instance.

5.5. Plugin: statistic

The `statistic` plugin provides classes to create statistics for the family mining process. The `GeneralStatistic` class provides methods to start and stop a timer for the parsing process. Furthermore, we have to set an `ApproachStatistic` class for the current approach, which we execute. The implementation of the `toString()` method prints the general statistic and the approach specific statistic to the console. The `GeneralStatistic` class is realized as a singleton class, since only one instance of the statistic is needed

The abstract `ApproachStatistic` provides methods to increment different counters, which are used to count the number of created `CompareElements` for states, regions, transitions, and state

charts and a number of other characteristics (e.g., the number of matching algorithm calls, or the number of decision wizard calls). Furthermore, the `ApproachStatistic` allows to add the models' sizes to the statistic by using the `addModelSize()` method, which stores the name of the passed model and the number of contained states, regions, and transitions. Different timers exist for the three phases of the family mining process (i.e., the `Comparing Phase`, the `Matching Phase`, and the `Mering Phase`), which can be started, stopped, and in the case of the comparison timer also paused and resumed.

Two methods exist, which are called when ambiguous elements were found during the *Matching Phase* (i.e., the `setAmbiguous()` method) and when they are resolved (i.e., the `setAmbiguousness-Resolved()` method). The `toString()` method for the `ApproachStatistic` prints the runtime of the current approach and also shows statistics about the compared state charts and the identified variability. Both statistics, the `GeneralStatistic` and the `ApproachStatistic` use the `Timer` class, which allows to start and stop a timer, get the timer's runtime and print the runtime as a formated string, which shows the hours, minutes, seconds, and milliseconds. Two implementations of the abstract `ApproachStatistic` exist (i.e., for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm), which are realized as singleton classes, because similar to the `GeneralStatistic` only one instance of the corresponding statistic is needed.

5.6. Plugin: compareengine

The `compareengine` plugin is the most complex plugin, because it contains all classes, which are needed to execute family mining on state charts. It contains multiple sub-packages, which group classes for different phases and tasks of the family mining process together. In order to give a small summary of each package and an overview of the package hierarchy used in the `compareengine` plugin, we consider Figure 5.3.

The most important class in the `compareengine` plugin is the abstract `CompareEngine` class, as it defines the methods that need to be realized in order to create a compare engine for family mining of state charts. In Figure 5.4, we present a simplified class diagram for this class. The first two abstract methods, which need to be implemented are the `initApproachStatistic()` method and the `initFactories()` method. The `initApproachStatistic()` method has to initialize the `statistic` variable, which is used to collect statistics about the family mining process. The `initFactories()` method is needed to initialize all factory variables. These factories are used in the course of the family mining process to execute the *Comparing Phase* (cf., Subsection 5.6.2), the *Matching Phase* (cf., Subsection 5.6.3), and the *Merging Phase* (cf., Subsection 5.6.4). These two initialization methods are called by the `initBehavior()` method, which gets the `Metric` (cf., Subsection 5.6.1) and the `DecisionWizard` (cf., Subsection 5.6.3) as parameters. The metric is used during the comparison of the elements to calculate their similarity and during the merging, since it defines the identified variability according to the calculated similarity. The decision wizard is needed during the matching in order to solve conflicts between ambiguous elements. Both elements are the only way how the family mining process can be influenced externally, without implementing a new version of one or multiple of the phases.

Any class, which extends the `CompareEngine` class, has to implement the four methods `compare()`, `match()` (this method exists in two versions), and `merge()`, which define how the phases have to be executed for the corresponding implementation. The execution of these three phases is started

- **compare**
  This package provides all classes, which are needed to compare state charts with each other.
  - **compareelement**
    This package contains classes, which are used to create different compare elements for state charts, states, regions, and transitions.
    - **adapted**
      This package contains the `compare` classes for the compare algorithm of the AdaptedFaMine algorithm.
    - **calculation**
      This package contains classes to store the similarity calculations during the comparison of state chart elements.
    - **improved**
      This package contains the `compare` classes for the compare algorithm of the ImprovedFaMine algorithm.
  - **handler**
    This package contains classes, which help to prevent loops when creating the compare elements during the *Comparing Phase*.
  - **multiple**
    This package contains classes, which are used to create all possible permutations of two lists of multiple elements with the same type (e.g., state actions, or transition labels), compare them, and return the best overall similarity.
- **helper**
  This package contains a helper class, which provides methods used by multiple classes in the compare engine.
- **match**
  This package provides all classes, which are needed to match a list of compare elements.
  - **decision**
    This package contains the classes, which are needed to realize a decision wizard to automatically select a compare element, when ambiguous elements are detected.
- **merge**
  This package contains all classes, which are needed to merge the results of the family mining.
- **metric**
  This package contains all classes, which are needed by the metric.
- **print**
  This package contains a class, which is used to print the results of the family mining process.

Figure 5.3.: Package hierarchy of the `compareengine` plugin

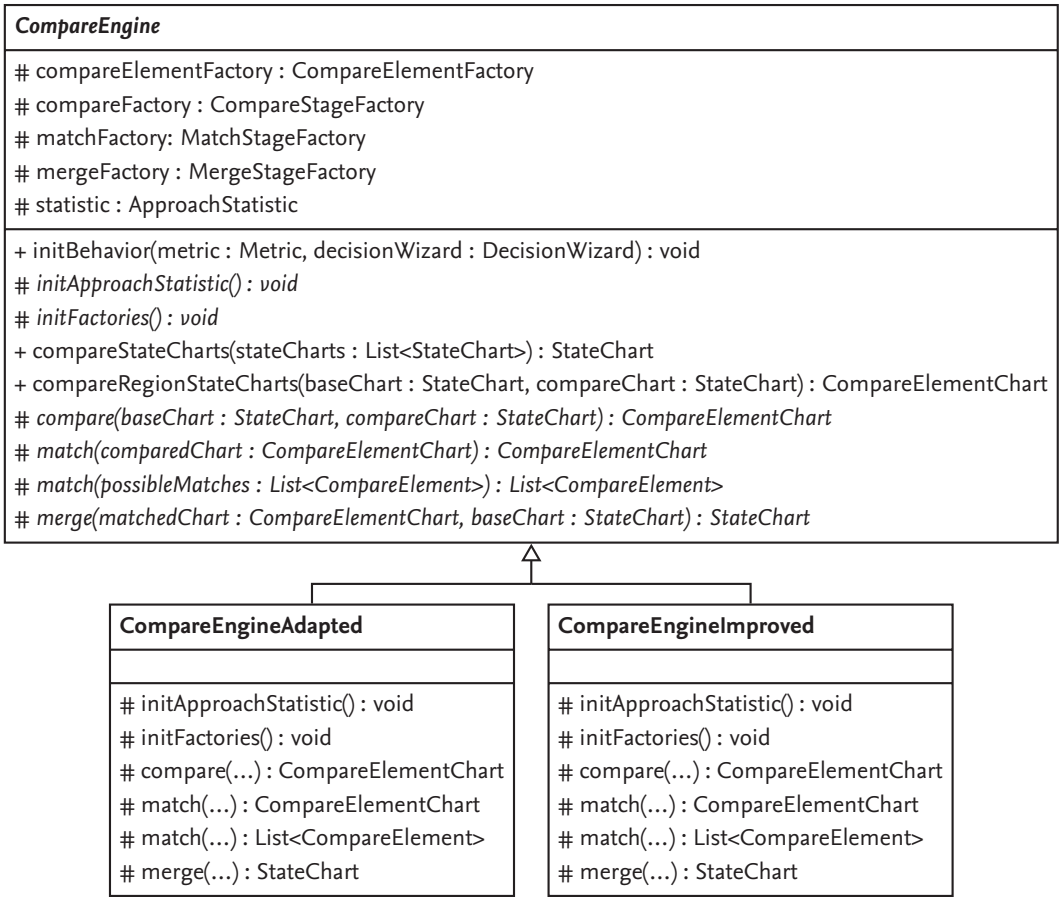


Figure 5.4.: Class diagram of the `CompareEngine`

by passing a list of `StateCharts` to the `compareStateCharts()` method, which calls the corresponding implementations for the three phases (i.e., the `compare()` method, the `match()` method, and the `merge()` method). The `compare()` method gets two `StateCharts` as parameters, which should be compared and returns a `CompareElementChart`, that contains the `CompareElements` for states and transitions, which were created during the comparison. This `CompareElementChart` is used as an input for the first `match()` method, which identifies distinct matches for the created `CompareElements`. The results of the matching are then passed to the `merge()` method, which also receives a copy of the base state chart and merges the results into this state chart and returns them as a `StateChart` object. If another state chart exists, that was not already compared and merged with the previous state charts, the result of the merging process is used as an input for the next execution, otherwise the resulting `StateChart` is returned. The second implementation of the `match()` method allows to find the distinct matches for a list of possible matches, which are not contained in a `CompareElementChart`; for example, to match a list of region compare elements, when identifying the correct combination of region compare elements. The `compareRegionStateCharts()` method is used to compare the contents of regions with each other, by transferring them to temporary `StateCharts` and recursively applying the *Comparing Phase* and *Matching Phase* to them.

The matched sub-states and sub-transitions from the regions are then stored in the compare element, which compares the regions' parents. Since *EMF* automatically manages the containment

of object references in lists[4], we use different copies of the models during all phases (created with the `EcoreUtil.copy()` method provided by *EMF*). Otherwise, the processing of such containment lists might be destructive, because adding an object reference from a containment list `A` to another list `B` automatically removes the object reference from list `A`. Consequently, list `A` might be modified unintentionally.

Currently, we implemented the two algorithms discussed in this thesis (i.e., the AdaptedFaMine algorithm and the ImprovedFaMine algorithm). The `CompareEngineAdapted` class implements the AdaptedFaMine algorithm and the `CompareEngineImproved` class implements the ImprovedFaMine algorithm, which we introduced in Section 4.6. Since this approach combines the comparison and the matching in one step (cf., Section 4.6) the implementation of the `match()` method for `CompareElementCharts` simply returns the `CompareElementChart`, which it gets as an input. Because of the space limitations in Figure 5.4, we refrained from displaying all parameters of the methods `compare()`, `match()`, and `merge()`.

In the following sections, we will further explain the implementation of the metric (cf., Subsection 5.6.1) and the three phases of the family mining process (cf., Subsection 5.6.2, Subsection 5.6.3, and Subsection 5.6.4).

5.6.1. Metric Implementation

The implementation of the metric consists of an abstract `Metric` class, which defines the structure, that has to be implemented, in order to realize a legal metric for the family mining process. If all needed methods are not implemented correctly, the `checkMetric()` method, which is called by the constructor of the abstract `Metric` class, throws an `IllegalMetricException`.

When extending the `Metric` class, all methods creating the weights for the different state chart elements have to be implemented correctly. In Listing 5.5, we present some example code, which shows a basic implementation for such a method. As we can see, a `HashMap` is created, which contains all elements, that need to be present for the corresponding part of the metric. In our example, we implement the method, which returns the weights for the *static* and *dynamic* parts of states. Static parts of a state do not contribute to its functionality (e.g., the name of a state), whereas dynamic parts influence its functionality (e.g., the state's actions). When adding the weights to the map, we use the constants defined by the abstract `Metric` class as key values (i.e., in this example `METRIC_STATE_STATIC` and `METRIC_STATE_DYNAMIC`). This enables the `checkMetric()` method to find the assigned weights and to check their validity. A list of all key values with short explanations can be found in Table A.2 in Section A.2. A table with all creation methods that need to implement these weights and the relation with the key values from Table A.2 can be found in Table A.3. All implementations of these methods, creating a map with values for the metric, have to make sure, that their assigned weights exactly sum up to 1.0. Otherwise, the `checkMetric()` method throws an `IllegalMetricException`.

As discussed in Section 4.4, the implementation of the `Metric` also has to provide the three methods `isMandatory()`, `isAlternative()`, and `isOptional()`. The input parameter for these methods is a compare element, whose similarity values has to be checked to return the requested status of the compare element. Consequently, these methods define the thresholds for the classification of the variability for the compare elements.

[4] http://www.eclipse.org/modeling/emf/

```
1 @Override
2 protected Map<String, Double> createStateWeightsMap() {
3   Map<String, Double> metric = new HashMap<>();
4   metric.put(METRIC_STATE_STATIC, 0.5);
5   metric.put(METRIC_STATE_DYNAMIC, 0.5);
6
7   return metric;
8 }
```

Listing 5.5.: Example for a creation method in the metric

Besides providing the weights for the different compared elements and methods to determine the variability of the created compare elements, the metric also holds a `StringSimilarityAlgorithm`, which should be used to compare strings with each other. The `StringSimilarityAlgorithm` is an abstract class which defines the abstract `stringSimilarity()` method. This method has to be implemented by extending classes. It gets two `Strings` and returns a double value between 0.0 and 1.0, indicating how similar the compared strings are. If the calculated value is below 0.0 or greater than 1.0, the method returns an `IllegalMetricException`. By implementing different algorithms to compare strings with each other, we can apply our approach to different use cases. For example, we can realize algorithms, which are more tolerant towards small differences between the compared strings (e.g., spelling mistakes) and which calculate a partial similarity between them, or others which do not tolerate these differences and strictly check for equality. All described classes can be found in the `metric` package.

5.6.2. Comparing Phase Implementation

The implementation for the *Comparing Phase* is contained in the `compare` package and consists of classes, realizing the compare algorithms, and classes, storing the results of the calculations and comparisons. The abstract `Compare` class defines the structure of compare stages. It contains a `compareStateCharts()` method, which is called with two state charts, that should be compared. This method initializes the algorithm and calls the abstract `generateCompareElements()` method. Extending classes should use the two lists `allPossibleMatchesState` and `allPossibleMatchesTransition` to store all possible matches for states and transitions, or directly store the matched states and transitions in the lists `optimalMatchesState` and `optimalMatchesTransition`.

As we can see in Figure 5.5, currently two implementations for compare algorithms exist, namely `CompareAdapted` for the AdaptedFaMine algorithm and `CompareImproved` for the ImprovedFaMine algorithm. These implementations for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm are realized as in the explanations in Section 4.2 and Section 4.6, consequently we do not go into much detail, but emphasize specialties.

Since we can compare multiple state charts with each other, we also have to include the variability, which was identified in previous comparisons. This only affects *alternative* elements, since we have to compare all elements from alternative groups with the new element from the new compare state chart. Here, we distinguish between the two cases in Figure 5.6:

1. The alternative group *contains* an element, which has 100% similarity compared to the element from the new compare state chart. For example, when comparing the alternative group in

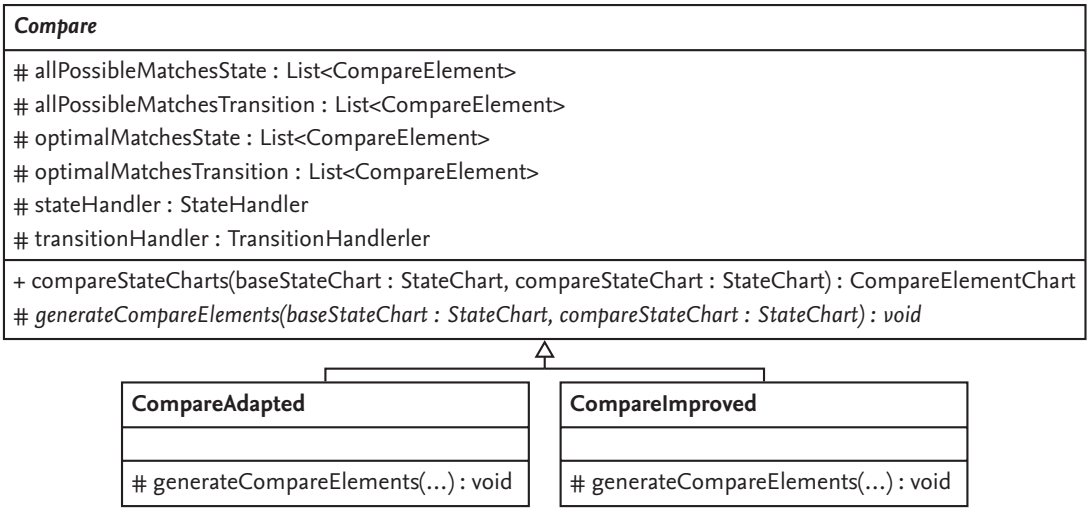


Figure 5.5.: Class diagram of the `Compare` implementation

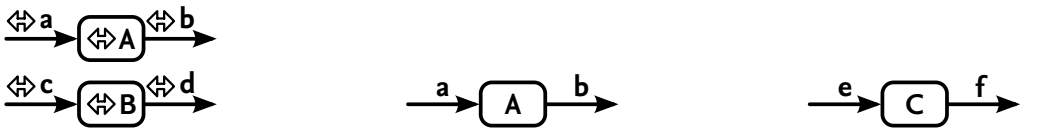
Figure 5.6a with Figure 5.6b from a new compare state chart, state `A` does not represent a new alternative, since it is the same as state `A` in Figure 5.6a.

2. The alternative group *does not contain* an element, which has 100% similarity compared to the element from the new compare state chart. For example, when comparing the alternative group in Figure 5.6a with Figure 5.6c from a new compare state chart, state `C` represents a new alternative, since it matches neither state `A`, nor state `B` in Figure 5.6a.

Our approach to handle these two cases is to compare any new element, which is compared to an element from an alternative group, with all elements from the corresponding alternative group. For example, when identifying that state `A` in Figure 5.6a is compared with state `C` from Figure 5.6c, we also compare it with state `A`. In order to prevent us from comparing the same element multiple times with the same alternative group, we keep track of all elements from the compare state charts, which we already compared with alternative groups. As the new element can only be matched with one element, we execute some preprocessing for the matching algorithm in order to reduce the number of compare elements it has to process.

After creating all compare elements for the new element compared to the alternative group, we iterate over them and select the element with the highest similarity. When identifying a *mandatory* element (i.e., no new alternative is created), we directly return the corresponding compare element and use this element. All other elements, that were ruled out are deleted afterwards and do not influence the matching process. As the ImprovedFaMine algorithm only uses the *Comparing Phase* and *Matching Phase* directly for transitions and matches states indirectly (cf., Section 4.6), we apply this technique only for transitions, whereas for the AdaptedFaMine algorithm it is applied for states *and* transitions.

The results of the comparisons have to be stored. Thus, we created the inheritance hierarchy in Figure 5.7. At the top, we created the `AbstractCompareElement` class, which allows to store a `ComparableStateChartElement` for the base element and the compare element. Two classes inherit from this class. First, the abstract `CompareElement` class, which adds the similarity of the compared elements, and overrides the `toString()` method to print the compared elements



(a) Alternative group between state A and state B

(b) Mandatory state A

(c) New alternative state C

Figure 5.6.: Example for the comparison of an alternative group to new elements

and their similarity to the console. This class also contains all methods, which were already realized by the current approach for block-based models to handle compare elements during the *Matching Phase* (e.g., to set the compare elements ambiguous). In addition, the abstract `init()` method should be implemented by inheriting classes to initialize their settings. Second, the concrete `CompareElementChart` class needs to be implemented, which does not need to store the similarity of the compared charts and is used to store the comparison of two state charts. It contains two lists to store all possible matches for states and transitions and also two lists for the results of the matching for these elements. These lists can only be accessed by using the corresponding getter and setter methods. Besides, the `CompareElementChart` class extends the `toString()` method to print its contents to the console. If the matching was not yet executed, the `toString()` method prints all possible matches and otherwise the matched results.

Three abstract classes inherit from the `CompareElement` class. The abstract `CompareElementState` class adds four methods to the `CompareElement`. The `isInitialStateCompareElement()` method returns whether the corresponding compare element compares two initial states, which is important for the *Merging Phase*, since there has to be exactly one initial state in a region and initial states cannot have variability. Consequently, they are always regarded as *mandatory*. The `isHierarchicalStateCompareElement()` and `isParallelStateCompareElement()` methods return whether the corresponding compare element compares hierarchical states or parallel states. These elements contain compare elements for the states' sub-regions, which can be accessed by the `getSubRegionCompareElements()` method. The `CompareElementState` class implements the algorithms to compare states with different hierarchies, which we discussed in Subsection 4.2.2.

The abstract `CompareElementRegion` class allows to compare regions with each other and stores the compare elements for its compared sub-contents (i.e., for the compared states and transitions). These can be accessed using the `getSubStateCompareElements()` and `getSubTransitionCompareElements()` methods, respectively. In order to compare two regions with each other, we create temporary state charts for these sub-contents and recursively call the compare and match algorithms by calling the `compareRegionStateCharts()` method of the `CompareEngine` class.

The abstract `CompareElementTransition` class compares transitions with each other (as discussed in Subsection 4.2.2) and stores the corresponding results. For all classes described to this point, there exist implementations for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm, which we do not further explain, since they use the implementation of the corresponding parent class and only differ in the initialization for the corresponding algorithm. All described abstract classes can be found in the `compareelement` package, together with their implementations for the AdaptedFaMine algorithm and the ImprovedFaMine algorithm. Because of the limited space available, we did not add these concrete sub-classes to Figure 5.7.

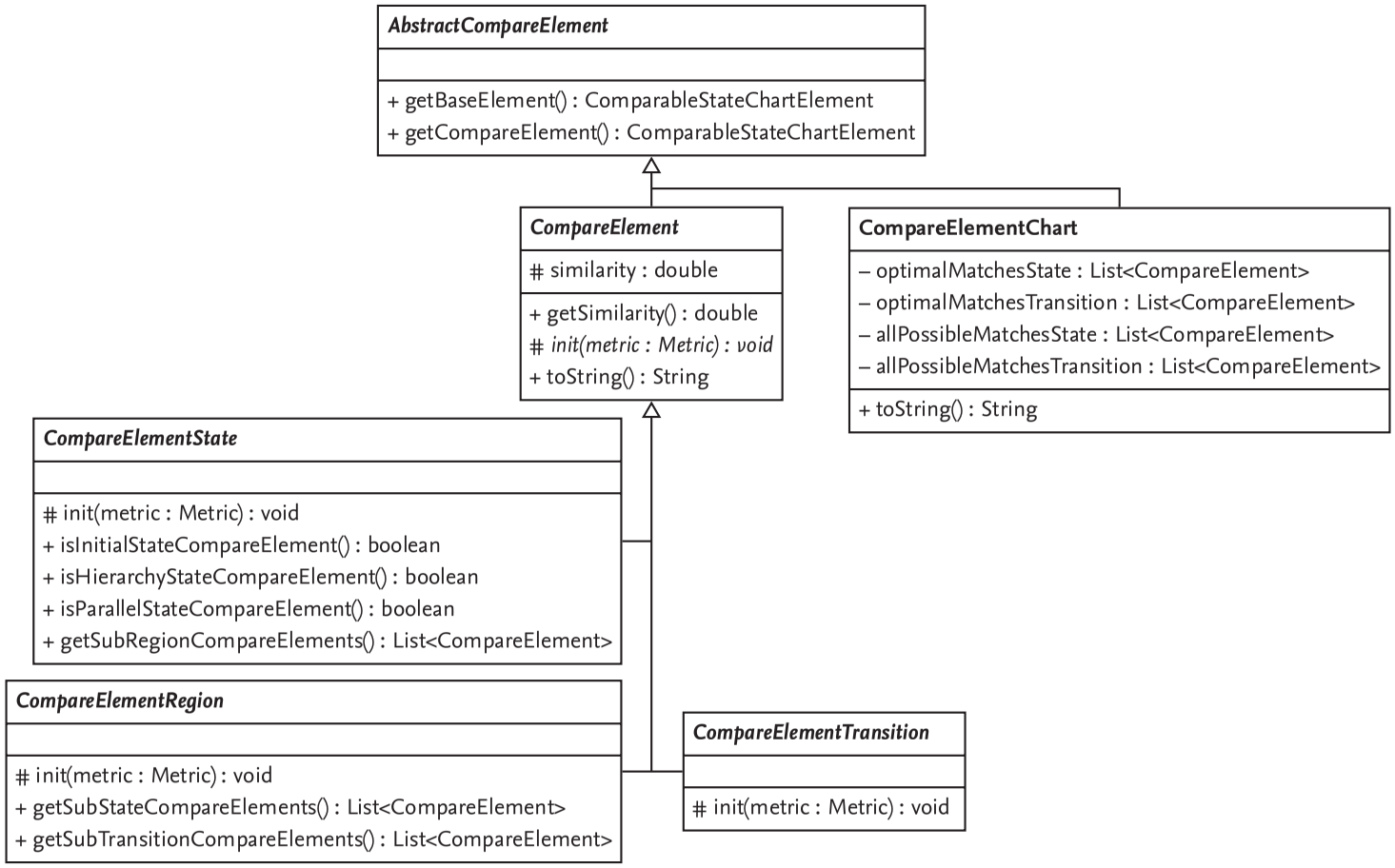


Figure 5.7.: Class diagram of the `CompareElement` implementation

```
1  public class StateStaticCalculation extends LinkedCalculation {
2
3    protected StateStaticCalculation(Metric metric)
4        throws CalculationAlreadyExistsException
5    {
6      super(metric, metric.getStateStaticElementsMap(), "State Static");
7    }
8
9    @Override
10   protected void init(Metric metric)
11       throws CalculationAlreadyExistsException
12   {
13     initCalculation(Metric.METRIC_STATE_STATIC_NAME);
14      [...]
15     initSubCalculation(Metric.METRIC_STATE_STATIC_NEIGHBOR,
16                        new StateNeighborhoodCalculation(metric));
17      [...]
18   }
19
20 }
```

Listing 5.6.: Example for an implementation of the `LinkedCalculation` class

In order to store the calculated similarity of compare elements in a more sophisticated way, we created the abstract `LinkedCalculation` class in the `calculation` package, which allows to store the intermediate results of the corresponding calculation. By using this class instead of a simple `double` value, we are able to better analyze the results of the calculation, since we can reproduce how the total similarity was calculated. Obviously, this way of storing the results is more complex, but it allows us to understand the behavior of the algorithms and eases adjustments of the metric.

In Listing 5.6, we present the implementation of the `LinkedCalculation` class, which is used to store the comparisons of the static attributes in states. In Line 6, we call the super constructor and pass the metric that should be used to compare the attributes of the states, the map containing the weights for the similarity calculation of the compare attributes, and the name of the current calculation, which is used when printing the calculation with all its sub-elements to the console. In order to extend the `LinkedCalculation` class, we also have to implement the `init()` method. This method initializes the values of the attributes, which are expected for the created calculation object. In Line 13, we show how a calculation for a simple `double` value has to be initialized. In this case the `initCalculation()` method gets the name of the attribute that should be initialized. If the attribute already exists, we throw an `CalculationAlreadyExistsException`, in order to prevent overriding the values of attributes, that were previously initialized. Otherwise, the value for the corresponding attribute is initialized to 0.0. As a `LinkedCalculation` can also contain sub-calculations for other complex calculations, we also have to initialize these sub-calculations. In Line 15, we show how such a sub-calculation is initialized. In this case, we pass the name of the attribute and the corresponding object (e.g., the calculation of the states' neighborhood in our example) to the `initSubCalculation()` method.

The overall similarity of the `LinkedCalculation` objects is not calculated until the `calculateOverallSimilarity()` method is called for the first time. This method iterates over the stored

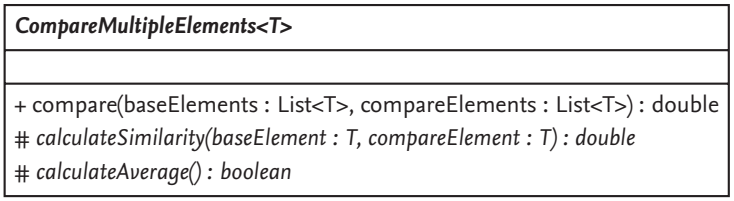


Figure 5.8.: Class diagram of the `CompareMultipleElements` implementation

attributes and their values and recursively calls the `calculateOverallSimilarity()` method for sub-calculations and multiplies all values with the corresponding weights defined by the map, which we initially passed to the super constructor. The calculated value is stored and is not re-calculated until the corresponding `LinkedCalculation` object has changed. Changes can only occur, when a value or a sub-calculation is initialized or value is changed by the `putCalculation()` method, which expects the name of the attribute, whose value should be changed, and the corresponding value.

Some of the comparisons for states and transitions need to compare multiple attributes of the same type (e.g., two lists of state actions from two states) and need to find the combination, which generates the highest similarity. In order to ease implementing such comparisons, we created the abstract generic `CompareMultipleElements` class in the `multiple` package, which is outlined in Figure 5.8. When calling the `compare()` method, this class compares every element from the first list with every element from the second list, and uses the abstract `calculateSimilarity()` method to calculate the similarity for the comparisons. After creating all comparisons, the class iterates over these results and for every iteration selects the comparison with the highest similarity. All other comparisons containing the same element from the first list or the second list are deleted from the list of comparisons. So, we maximize the total similarity of the compared elements. Before returning the total similarity, the `compare()` method uses the abstract `calculateAverage()` method to check whether the similarity should be divided by the number of compared elements. For the `CompareMultipleElements` class, currently there exist implementations for `Actions`, `Strings`, `TransitionLabel`, and neighbor `States`.

In order to have a central class which creates and manages compare elements, we created the abstract `CompareElementFactory` class in Figure 5.9. It contains the abstract `createCompareChart()` method, which should be implemented by inheriting classes and allows to create a new `CompareElementChart` to store the results for the comparison of two state charts. Besides, the `createCompareElement()` method exists which allows to compare two `ComparableStateChartElements` and uses the `CompareElementPool` class to manage the created compare elements. The abstract `CompareElementPool` provides the `getCompareElement()` method, which gets two `ComparableStateChartElements` as parameters and returns the corresponding `CompareElement`. First, the method checks whether both elements are not `null` and of the same type (e.g., two states) and, thus, are comparable. Otherwise, the method throws an `IllegalCompareException` or `BothComparedElementsNullException`, respectively. If the elements are comparable, the method first calls the `isAlreadyContained()` method and checks, whether a compare element already exists for this comparison. In this case, the corresponding compare element is returned, otherwise, it is created and afterwards returned. By using this `CompareElementPool`, we reduce the number of created

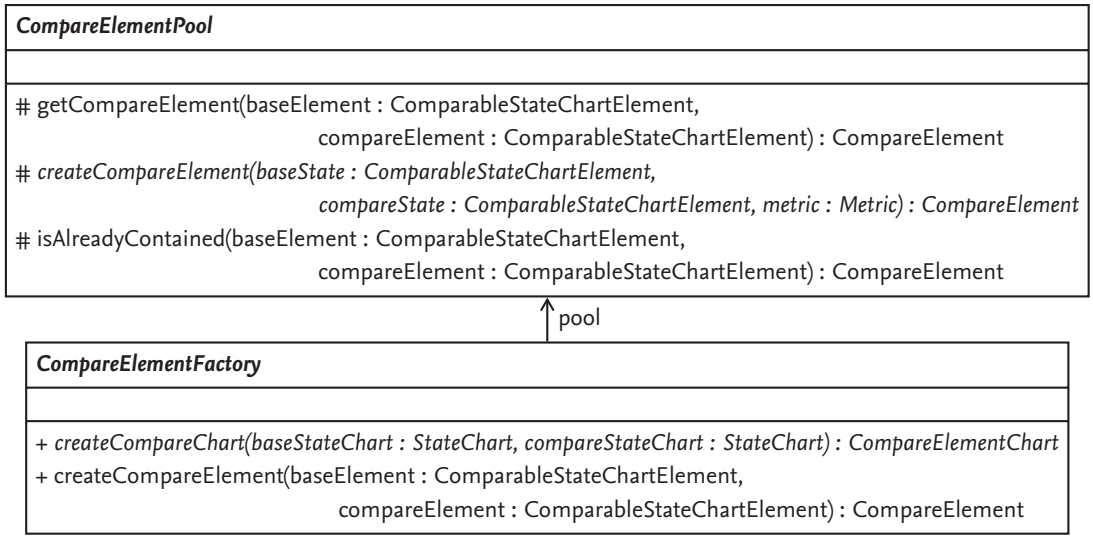


Figure 5.9.: Class diagram of the `CompareElementFactory` and `CompareElementPool` implementation

`CompareElements`, because they are reused during the comparison. For both approaches (i.e., the AdaptedFaMine algorithm and the ImprovedFaMine algorithm), there exist implementations for the described `CompareElementFactory` and `CompareElementPool` classes, which are realized as singleton implementations (i.e., they have a `getInstance()` method returning the only instance of the corresponding class).

The `Printing` class in the `print` package provides the static `printMergedResult()` method, which allows to print the results of the family mining process to the debug console (i.e., the console of the Eclipse runtime instance). This class is a replacement for a proper family model exporter, since we refrained from implementing such an exporter as discussed in Section 4.5. Mandatory elements are represented by exclamation marks (i.e., "`!`"), alternatives are represented by double arrows (i.e., "<=>"), and optional elements are indicated by question marks (i.e., "?").

In Listing 5.7, we present the results of a comparison which were created using the `printMergedResult()` method. As we can see, we identified that both models contain a mandatory initial state `Initial` and a mandatory state `State1`. All transitions are printed below their source state using an indentation of one tab in relation to their source state. For example, the transition in Line 4 starts at the state `Initial` and, consequently, it is indented. Any regions inside of states are also indented by one tab in relation to their parent state and the contents of these regions are printed below the region heading (e.g., for the region `state_1` in Line 6). Alternative transitions and states are added to groups (e.g., `Group 0` in Line 8) and are printed below their group heading with one tab indentation in relation to this heading. And finally, optional elements are represented by a question mark, as, for example, the region `state_2` in Line 19.

5.6.3. Matching Phase Implementation

The implementation for the *Matching Phase* is contained in the `match` package. As we can see in Figure 5.10, it basically consists of the abstract `Match` class and its sub-classes for the two approaches. The abstract `Match` class provides the abstract `init()` method which needs to be used by extending classes to initialize the `decisionWizard`, the `factory`, and the `statistic`, used during the

```
1  ## model1_model2_merge ##
2  ! ## Region: RootRegion ##
3  ! Initial
4    ! Source: Initial - Initial - Target: State1
5  ! State1
6    ! ## Region: state_1 ##
7    ! Initial
8    ## Group: 0 ##
9      <=> Source: Initial - Initial - Target: State3a
10     <=> Source: Initial - Initial - Target: State3b
11   ## Group: 1 ##
12     <=> State4a
13     <=> State4b
14   ## Group: 2 ##
15     <=> State3a
16       <=> Source: State3a - 1:  - Target: State4a
17     <=> State3b
18       <=> Source: State3b - 2:  - Target: State4b
19   ? ## Region: state_2 ##
20   ! Initial
21     ! Source: Initial - Initial - Target: State2
22   ! State2
```

Listing 5.7.: Example for a family mining result printed to the console

matching. The `matchCompareElements()` method starts the matching algorithm and gets the list of `CompareElements` that should be matched. Since all state charts elements, which we want to compare with the family mining approach are realized as `ComparableStateChartElements`, we adapted the current matching algorithm from `SimulinkBlocks` (i.e., the meta-model class from the current block-based approach) to `ComparableStateChartElements`. This allows us to match compare elements which contain elements extending the `ComparableStateChartElements` class. All algorithms used in the abstract `Match` class are the exact adaptations of the current implementation to `ComparableStateChartElements`, consequently, we do not explain them in further detail.

For both approaches (i.e., the AdaptedFaMine algorithm and the ImprovedFaMine algorithm), there exist implementations of the abstract `Match` class which only differ in the approach-dependent `init()` method.

As previously described, the `Match` class needs to be initialized with a `DecisionWizard`. The abstract `DecisionWizard` class provides methods to implement a decision wizard that allows to manually or automatically solve conflicts, which occur because of ambiguous compare elements during the `Matching Phase`. The `DecisionWizard` class can be found in the `decision` package and is adapted directly from the current approach, thus, we do not explain it in further detail.

5.6.4. Merging Phase Implementation

The implementation for the *Merging Phase* can be found in the `merge` package. This implementation basically realizes the steps described in Section 4.4. Consequently, we do not explain every step in much detail, but only concentrate on complex parts of the implementation, which we needed to realize these ideas. As we previously explained, *EMF* automatically manages containments of

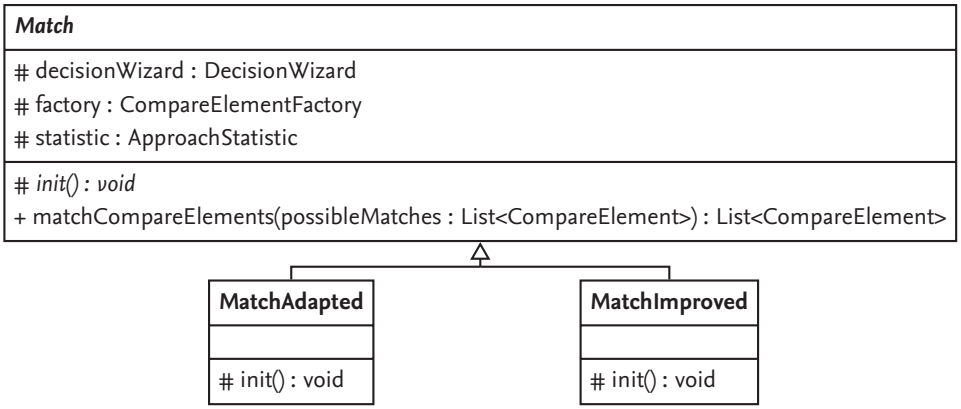


Figure 5.10.: Class diagram of the `Match` implementation

lists and, thus, we might unintentionally modify processed models. Consequently, we use different copies of the compared state charts during all phases of the implementation in order to prevent this behavior. During the creation of the compare elements, we use a different copy of the currently processed base state chart than for the merging process at the end of the family mining process. Hence, we have to find a way to map between the different object instances, because, for each of the base state chart elements, there exist two object copies (i.e., one copy in the state chart used for the *Comparing Phase* and one in the copy for the *Merging Phase*).

For this purpose, we created the `MergeSharedResources` class in Figure 5.11, which contains three maps (i.e., the `stateMap`, the `transitionMap`, and the `mandatoryCompareStateMap`), that manage the elements contained in the final state chart. These maps get `Strings` as keys and store the corresponding elements (i.e., states and transitions, respectively) from the state chart copy used for the merging process as values. In order to store these elements unambiguously in the map, we use the unique ids assigned to these elements as key values (cf., Section 3.4). Consequently, it is important that we assign truly unique ids to all element instances of the state chart (e.g., by using *Universally Unique Identifiers (UUIDs)*). For each of the maps, there exist methods to add elements, to check whether an element with a given id is already contained in the final state chart (i.e., in the corresponding map), and to get an element with a given id from the corresponding map. By passing the unique id for a state or transition object from the state chart copy, used for the comparing process to these methods, we can now request the corresponding element from the state chart copy used for the merging process. The returned object is used to merge the results correctly to the final state chart.

The `stateMap` and the `transitionMap` store all states and transitions which are currently contained in the final state chart. The `mandatoryCompareStateMap` is used during the merging process to manage *mandatory compare state chart states*. These states are identified to be mandatory compared in the base state chart and, thus, are not merged into the final state chart. When merging a transition, which has such a state as a source or target state, we cannot find it in the map of states currently contained in the final state chart, because it was not merged. Consequently, we would not be able to merge this transition into the final state chart, because transitions always need a legal source and target state. In order to solve this issue, we introduced the `mandatoryCompareStateMap`. When we encounter such a *mandatory compare state chart state*, we add the state to this additional map using

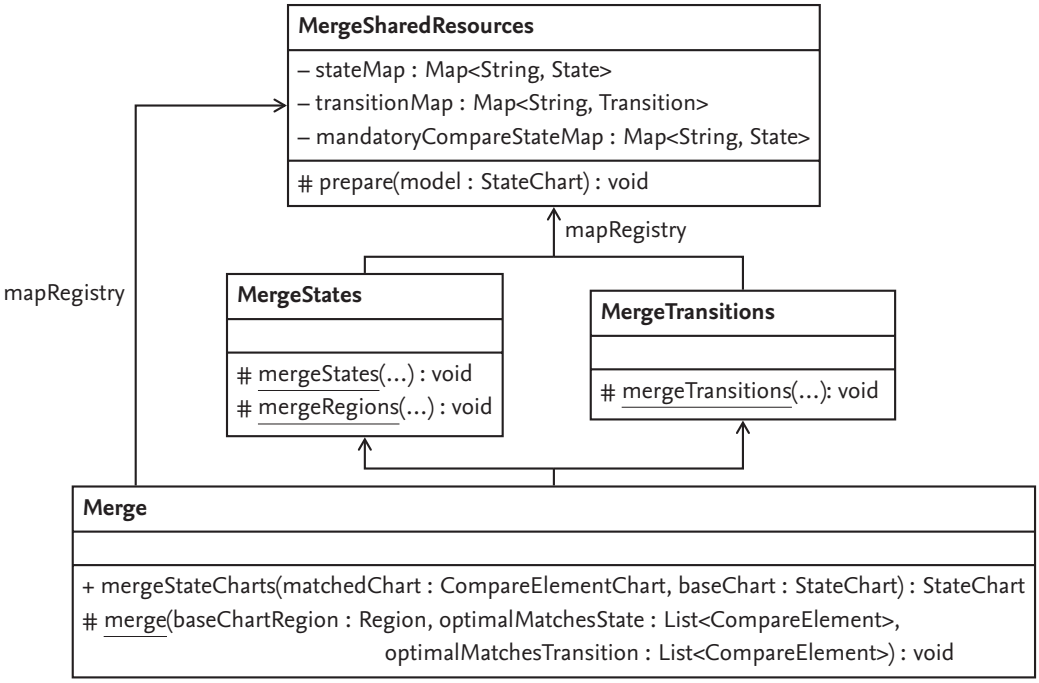
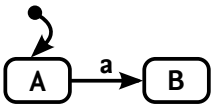


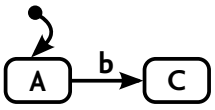
Figure 5.11.: Class diagram of the `Merge` implementation

its unique id as the key, but with its corresponding mandatory counter part in the base state chart (i.e., the state, which was identified to be the same compared to this state) as a value. For example, when comparing the two models in Figure 5.12, we first merge the corresponding states. During this merging process, we identify that both states `A` are the same and do not merge state `A` from the compare state chart into the final state chart, but add it to the `mandatoryCompareStateMap` using its unique id as a key and state `A` from the base state chart as a value. Furthermore, state `B` and `C` are identified as alternatives to each other and state `C` is merged into the final state chart accordingly. The next step is to merge transition `b` into the final state chart. When requesting the object for state `A` from the `stateMap` in the `MergeSharedResources` class, we receive `null` as a result. Consequently, we request the same object again, but in the `mandatoryCompareStateMap`, which returns the correct object for state `A` (i.e., the state `A` from the base state chart, which was already contained in the final state chart). Finally, we can merge transition `b` correctly as an alternative to transition `a` into the final state chart. This example shows that it is important to correctly add all merged states and transitions to the corresponding maps in the `MergeSharedResources` class, in order keep track of merged elements.

As we can see in Figure 5.11, the merging process is started by using the `Merge` class. It provides the `mergeStateCharts()` method, which expects the results of the *Matching Phase* (i.e., the



(a) Base state chart



(b) Compare state chart

Figure 5.12.: Example for *mandatory compare state chart states*

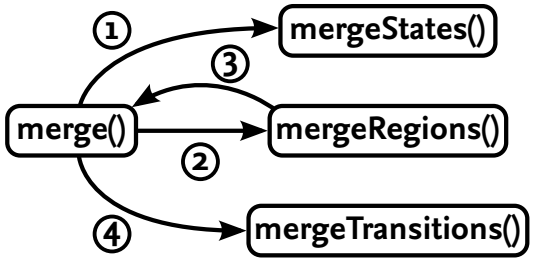


Figure 5.13.: Method calls during the merging of state charts

`CompareElementChart` containing these results) and the `StateChart`, which should later contain the merged results. By calling the `prepare()` method in the `MergeSharedResources` class with the `StateChart` copy, the `mergeStateCharts()` method initializes the corresponding maps for states and transitions. Next, the `mergeStateCharts()` method calls the `merge()` method, which uses the `MergeStates` and `MergeTransitions` classes to merge the given states and transitions into the base state chart region.

In Figure 5.13, we present the basic method calls during the merging process. When calling the `merge()` method in the `Merge` class, we start to process the list of optimal state matches. The first step is to call the static `mergeStates()` method in the `MergeStates` class (i.e., we follow 1 in Figure 5.13) for each of these states. This method expects as parameters the `Region`, which should contain the merged states, the `CompareElementState` that contains the compared states, and the `State` object, which is returned by the `statesMap` from the `MergeSharedResources` class for the corresponding base state in the compare element. This `State` object can only be `null`, when the compare element contains an optional element from the compare state chart.

After merging the state according to the steps described in Section 4.4 and adding the merged elements correctly to the corresponding maps in the `MergeSharedResources` class, the `merge()` method checks, whether the merged state is a hierarchical or parallel state. If the state is *not* hierarchical or parallel, the `mergeTransitions()` method from the `MergeTransition` class is used to merge the transitions into the final state chart (i.e., we follow 4 in Figure 5.13).

Similar to the `mergeStates()` method, this method expects as parameters the `Region`, which should contain the merged transitions, the `CompareElementTransition` that contains the compared transitions, and the `Transition` object, which is returned by the `transitionsMap` from the `MergeSharedResources` class for the corresponding base transition in the compare element. This `Transition` object can only be `null` when the compare element contains an optional element from the compare state chart. Before merging the transitions, the `mergeTransitions()` method requests the corresponding source and target states (i.e., from the `statesMap` and the `mandatoryCompareStateMap`, respectively) and uses them according to the steps described in Section 4.4. If the state is identified to be hierarchical or parallel, the `merge()` method calls the static `mergeRegions()` method in the `MergeStates` class to merge all sub-region compare elements into this state (i.e., we follow 2 in Figure 5.13). This method expects the `State`, which should contain these regions and the list of `CompareElementRegion` objects. After merging these sub-region elements into the final state chart, this method recursively calls the `merge()` method from the `Merge` class for each merged sub-region, in order to merge all sub-contents (i.e., states and transitions) into the final state chart (i.e., we follow 3 in Figure 5.13).

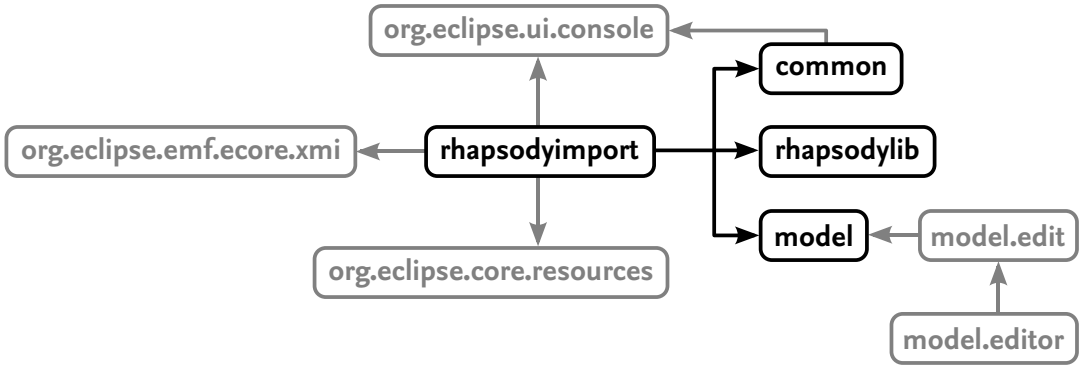


Figure 5.14.: Dependencies of the created Eclipse plugins for the *IBM Rational Rhapsody* importer

5.7. Implementation of the IBM Rational Rhapsody Importer

As the state charts of the case study, which we use for the evaluation in Chapter 6, were created with *IBM Rational Rhapsody*, we had to find a solution to import these state charts into our internal EMF meta-model. *IBM Rational Rhapsody* provides a Java API, in order to open models, access their contents, and use them for different purposes. We realized a *IBM Rational Rhapsody importer* using this library, in order to import `*.rpy` files (i.e., *IBM Rational Rhapsody* files) containing state charts to our internal EMF meta-model, and export the created representations to `*.statechart` files, which can be used for the evaluation of the family mining approach for state charts. We could have realized the importer without exporting the internal EMF representation to `*.statechart` files and directly using the imported information for the family mining. Instead, we decided to create the files, because this way, we have to process the *IBM Rational Rhapsody* files only once and can reuse the exported `*.statechart` files. Besides, we can run the family mining approach with `*.statechart` files on computers without an *IBM Rational Rhapsody* installation, which is not possible when importing the state charts for each execution, since the importer needs this installation to process the `*.rpy` files.

Similar to the family mining implementation described in Section 5.1, we realized this importer by using the Java Development Kit (JDK) in version 8[5] creating Eclipse RCP plugins for Eclipse Luna[6]. In Figure 5.14, we present the created architecture for the *IBM Rational Rhapsody importer* with all dependencies between the plugins.

The basic purpose of the plugins shown in Figure 5.14 is described in the following overview. A directed transition between two plugins indicates, that the plugin, in which direction the arrow is pointing, is a dependency of the other plugin (e.g., the `rhapsodyimport` depends on the `rhapsodylib`).

rhapsodylib

This plugin contains the *IBM Rational Rhapsody* Java API, which is needed to execute the import process from *IBM Rational Rhapsody* to our internal EMF meta-model. The details of this plugin are explained in Subsection 5.7.1.

[5] https://www.oracle.com/java/
[6] https://www.eclipse.org/

rhapsodyimport
This plugin extends the `org.eclipse.ui.commands`, `org.eclipse.ui.handlers`, and `org.eclipse.ui.menus` extension points and creates a new menu in the menu bar, which contains buttons to trigger the import and processing of the `*.rpy` files and the export. The details of this plugin are further explained in Subsection 5.7.2.

model
This is the same plugin, already explained in detail in Section 5.3. It contains the *meta-model* described in Section 3.4, which was created using the Eclipse Modeling Framework (EMF)[7].

model.edit / model.editor
These plugins are not directly needed for the *IBM Rational Rhapsody importer*, but provide methods and classes to display and edit XMI files in the `*.statechart` format, which is helpful to check the results of the import and export process.

common
This plugin contains classes, which are used in different other plugins. The only reason, why this plugin is needed, is the console implementation, which can be used for debugging purposes. All details of this plugin are explained in Section 5.4.

org.eclipse.core.resources
This plugin is used to access certain resource handling classes of Eclipse, which allow to access the files selected in the *Package Explorer* or *Project Explorer* of Eclipse.

org.eclipse.emf.ecore.xmi
This plugin is used to create *XML Metadata Interchange (XMI)* files, a special *Extensible Markup Language (XML)* format, which can be used to store models created with the defined meta-model.

org.eclipse.ui.console
This plugin is used to realize the debug console provided by the `common` plugin.

5.7.1. Plugin: rhapsodylib

The `rhapsodylib` plugin provides the *IBM Rational Rhapsody* Java API, which is contained in the *IBM Rational Rhapsody* installation folder in `Share/JavaAPI`. As we use *IBM Rational Rhapsody 8.0.6* under *Windows* in *64bit* this folder contains the `rhapsody.jar`, the `rhapsody.dll` for *Windows 64bit*, and a `WIN32` folder, containing the `rhapsody.dll` for *Windows 32bit*. In order to use these files in an Eclipse plugin, we copied the needed libraries from the *IBM Rational Rhapsody* installation folder and created the `rhapsodylib` plugin, which encapsulates these libraries for the access to *IBM Rational Rhapsody*. The plugin can be added as a dependency to any Eclipse plugin, in order to use them.

The plugin contains a `lib` folder with the `rhapsody.jar` file and a another folder `dll`, which contains two additional folders `32bit` and `64bit`. These folders include the native code, which is needed to execute the Java API (i.e., the `rhapsody.dll` files for the corresponding computer architecture). At runtime the plugin exports the `com.telelogic.rhapsody.core` package, which

[7] http://www.eclipse.org/modeling/emf/

```
1  [...]
2  Bundle-NativeCode:   lib/dll/32bit/rhapsody.dll; osname=win32; processor=x86,
3  lib/dll/64bit/rhapsody.dll; osname=win32; processor=x86_64
4  Bundle-ClassPath:    .,
5  lib/rhapsody.jar
6  [...]
```

Listing 5.8.: Adding the necessary libraries to the `MANIFEST.MF` file

is provided by the `rhapsody.jar` file, and, thus, allows to access and use the Java classes contained in this package of the `rhapsody.jar`.

In Listing 5.8, we present an excerpt of the `MANIFEST.MF` file in the `rhapsodylib` plugin, which shows how the native code in form of the `*.dll` files for the corresponding architectures is added to the plugin, and how the classes in form of the `*.jar` file are added to the class path. In Line 2 and Line 3, we add the native code for *Windows 32bit* and *Windows 64bit*, respectively. For both versions, we have to use `win32` for the `osname` keyword, but define different `processor` versions (i.e., `x86` and `x86_64`, respectively). Parts of this definition (e.g., the path for the `*.dll` and the `osname`) are separated by semicolons and multiple definitions are separated by commas (e.g., for the two computer architectures). In Line 4 and Line 5, we include the `rhapsody.jar` file in the class path of the plugin. We first add a dot to the class path (cf., Line 4). This dot defines, that all classes, which are part of the plugin's source folder, are considered during execution. In Line 5, we add the `rhapsody.jar` to the class path, using the same notation as for the native code and separating it with a comma from the previous definition.

In order to use our `rhapsodylib` plugin, we have to install *IBM Rational Rhapsody 8.0.6* under *Windows*, because without a legal installation of *IBM Rational Rhapsody* the plugin will not work. If *IBM Rational Rhapsody* is only available in another version or for *Linux*, the corresponding libraries have to be copied to the right location in the `lib` folder and, depending on the differences (e.g., `*.dll` files under *Windows* versus `*.so` files under *Linux*), the path in the `MANIFEST.MF` file has to be modified. When all preconditions are met (i.e., correct installation and correct settings in the `MANIFEST.MF` file), we can include the `rhapsodylib` plugin as a dependency in a new plugin and use its functionality. Thus, we can use all classes, which are included in the `rhapsody.jar` file and also look up their documentation in the Javadoc files in the `Doc/java_api` folder of the *IBM Rational Rhapsody* installation or on the corresponding website[8].

5.7.2. Plugin: rhapsodyimport

The `rhapsodyimport` plugin provides classes to import state charts from *IBM Rational Rhapsody* and to store them as images files, or in our internal EMF meta-model representation, which is exported as a `*.statechart` file.

The plugin extends the `org.eclipse.ui.commands`, the `org.eclipse.ui.handlers`, and the `org.eclipse.ui.menus` extension points to create a new menu bar entry, which is called "IBM Rational Rhapsody Import". The menu contains the "Export to XMI" command, which exports the imported state charts to `*.statechart` files, and a sub-menu, which contains entries to export the

[8] http://pic.dhe.ibm.com/infocenter/rhaphlp/v8/topic/com.ibm.rhp.api.doc/topics/rhp_r_ext_using_rhapsody_api.html (Status September 2014)

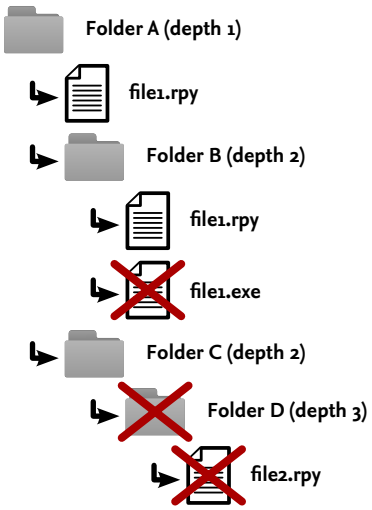


Figure 5.15.: Example for the collection of `*.rpy` files in the *IBM Rational Rhapsody* importer

imported state charts to `*.bmp`, `*.emf`, `*.jpg`, `*.jpeg`, and `*.tiff` image files. For each of these export possibilities exists a corresponding handler class in the inheritance hierarchy in Figure 5.16. Similar to the `gui` plugin for the family mining implementation (cf., Section 5.2), we extend the `AbstractHandler` class with the abstract `ExportHandler` class and override the `execute()` method to start the import and export process. The overridden method opens a wizard, which allows to select a folder, which should be searched for `*.rpy` files (i.e., *IBM Rational Rhapsody* files) and a folder depth, which defines the maximum search depth, when recursively calling the search algorithm for found folders.

In Figure 5.15, we present an example, how the files are collected. In this example, the user selected `Folder A` as the start folder for collecting *IBM Rational Rhapsody* files and a maximum search depth of two. The collection algorithm starts at depth one (i.e., in our example `Folder A`) and collects all files (i.e., in our example `file1.rpy`). If the algorithm finds another folder inside the current folder and it did not reach the maximum search depth, it recursively calls itself for the corresponding folders (i.e., in our example `Folder B` and `Folder C`). Any file, which does not have the file ending `*.rpy` is not collected for further processing (i.e., in our example `file1.exe`). Folders and files, which are at a level below the maximum search depth are not considered for the collection of files (i.e., in our example `Folder D` and `file2.rpy`). In order to simplify the example in Figure 5.15, we neglected all other project files, which are needed by *IBM Rational Rhapsody* to open a `*.rpy` file. Of course, these files are still needed, in order to open and process the found `*.rpy` files.

The `processFile()` method needs to be implemented by any concrete classes, in order to define how the found files should be processed. In Figure 5.16, we can see, that this method is implemented by the `XmiHandler` class and the `ImageHandler` class. The `XmiHandler` class uses the `Rhapsody2Xmi` class to import the found `*.rpy` files from *IBM Rational Rhapsody* into our internal EMF meta-model and to export them, using the code explained in Section 5.3, to `*.statechart` files. The `ImageHandler` on the other hand imports the found `*.rpy` files and exports them using the `Rhapsody2Image` class, which allows to export from *IBM Rational Rhapsody* to its five supported image types (i.e. `*.bmp`, `*.emf`, `*.jpg`, `*.jpeg`, and `*.tiff`). Each of the handlers, extending the `ImageHandler` defines one of these possibilities and exports to one of the corresponding types.

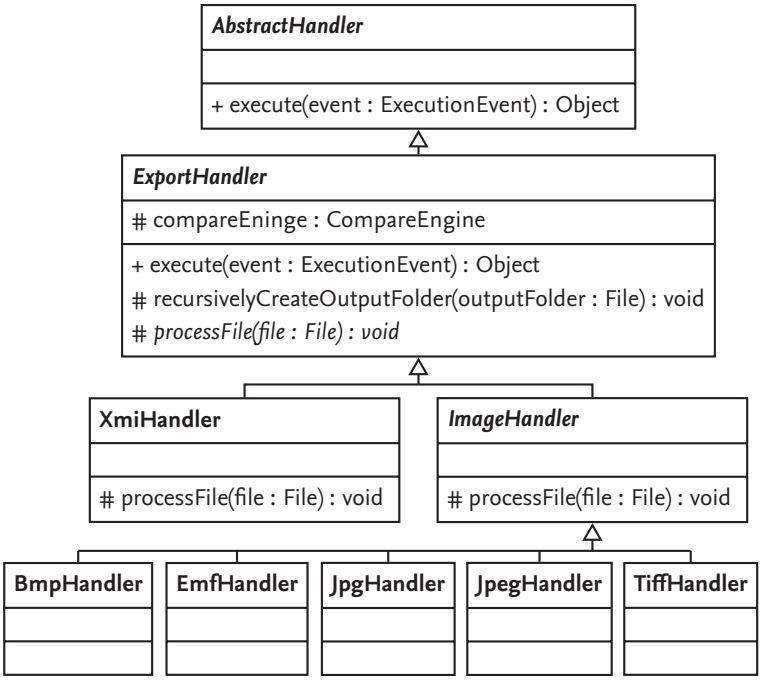


Figure 5.16.: Class diagram of the `ExportHandler` implementation

Both implementations of the `processFile()` method execute the `recursivelyCreateOutputFolder()` method for the currently processed file. This method checks if the output folder defined for the corresponding handler exists (i.e., `xmi_output` for the `XmiHandler` and `image_output` for the `ImageHandler`) and if it contains the folder hierarchy for the currently processed file. Otherwise, this hierarchy is created. For the example, in Figure 5.15 this would mean, that inside `Folder A` an output folder `xmi_output` is created, which contains `file1.statechart`. Furthermore, folder `Folder B` is created inside the `xmi_output` folder, which contains the other `file1.statechart`. As we can see, files with the same name are no problem during export, since they are created at their corresponding place in the output folder and are not overwritten by further files with the same name.

The `ExportHandler` class uses the `RhapsodyControl` class, which is realized as a singleton implementation and allows to open *IBM Rational Rhapsody* instances, open projects contained in `*.rpy` files, process these files, and close these files and the opened *IBM Rational Rhapsody* instance.

5.8. Summary

In this chapter, we explained the implementation of the two algorithms (i.e., the AdaptedFaMine algorithm and the ImprovedFaMine algorithm) explained in Chapter 4. Thereby, we concentrated on the most complex classes (e.g., the classes for the *Comparing Phase* and the *Merging Phase*) and did not explain classes in much detail, which are directly adapted from the current family mining implementation for block-based models (e.g., the classes for the *Matching Phase*). Furthermore, we explained the implementation of the tool, which we use to transfer state charts from *IBM Rational Rhapsody* into our internal meta-model. This tool is used during the evaluation in Chapter 6 to import state charts from the case study, which we use during the evaluation of our family mining implementation.

6 Evaluation

In this chapter, we evaluate our implementation of both algorithms, the AdaptedFaMine algorithm and the ImprovedFaMine algorithm. In Section 6.1, we describe the *BCS SPL case study*, which we use to evaluate our implementation. As previously described, we need different settings to execute the family mining for state charts (i.e., a metric, a string comparison algorithm, variability thresholds, and a decision wizard), which we introduce for our evaluation in Section 6.2. In Section 6.3, we present the research questions, which we consider during our evaluation of our current implementation. After selecting the *cases* (i.e., sets of state charts, which are compared to each other) for the evaluation in Section 6.4, we present the corresponding results in Section 6.5. These results are discussed in Section 6.6. Possible threats to validity, which might limit the universal validity, are explained in Section 6.7.

6.1. The Body Comfort System Case Study

The *BCS case study* was originally developed by Müller et al. [13] in cooperation with industrial partners from the automotive sector. This case study models the functionality of the body comfort system in a car, which contains different comfort features including electrically powered windows with finger protection, electrically adjustable and heatable exterior mirrors, and an alarm system with interior monitoring.

By decomposing the existing *BCS case study* into variable parts, Oster et al. [16] created the *BCS SPL case study*, which consists of *27 features* including alternative features (e.g., automatic windows, which only require the user to push the button once, or manual windows, which require the user to push the button until the window is closed or opened, respectively), optional features (e.g. the exterior mirrors can be heatable), and several features, which *require* other features in order to be executable (e.g., the status LED for the heatable exterior mirrors requires the selection of the heatable feature) [16, 25]. The *BCS SPL case study* allows to create 11,616 valid product variants and we will focus on the 17 product variants, which were created by Oster et al. [16] for their work on pairwise subset testing [16, 25] and the additional *core product* defined by Lity [10]. This core product represents the product variant, which is the identified basis for all 17 products defined by Oster et al. [16, 25] and contains all features common to these variants.

6.1.1. Basic Structure of the BCS SPL Case Study

In Figure 6.1, we present the feature diagram for the *BCS SPL case study*. These features allow to configure the *architecture* of the *BCS SPL case study* [11].

In order to better understand the *BCS SPL case study* and the meaning of the different features and their dependencies during the evaluation of our approach, we present the following list with short explanations of the different features and to introduce abbreviations for these features. The list is alphabetically ordered to ease searching for certain features.

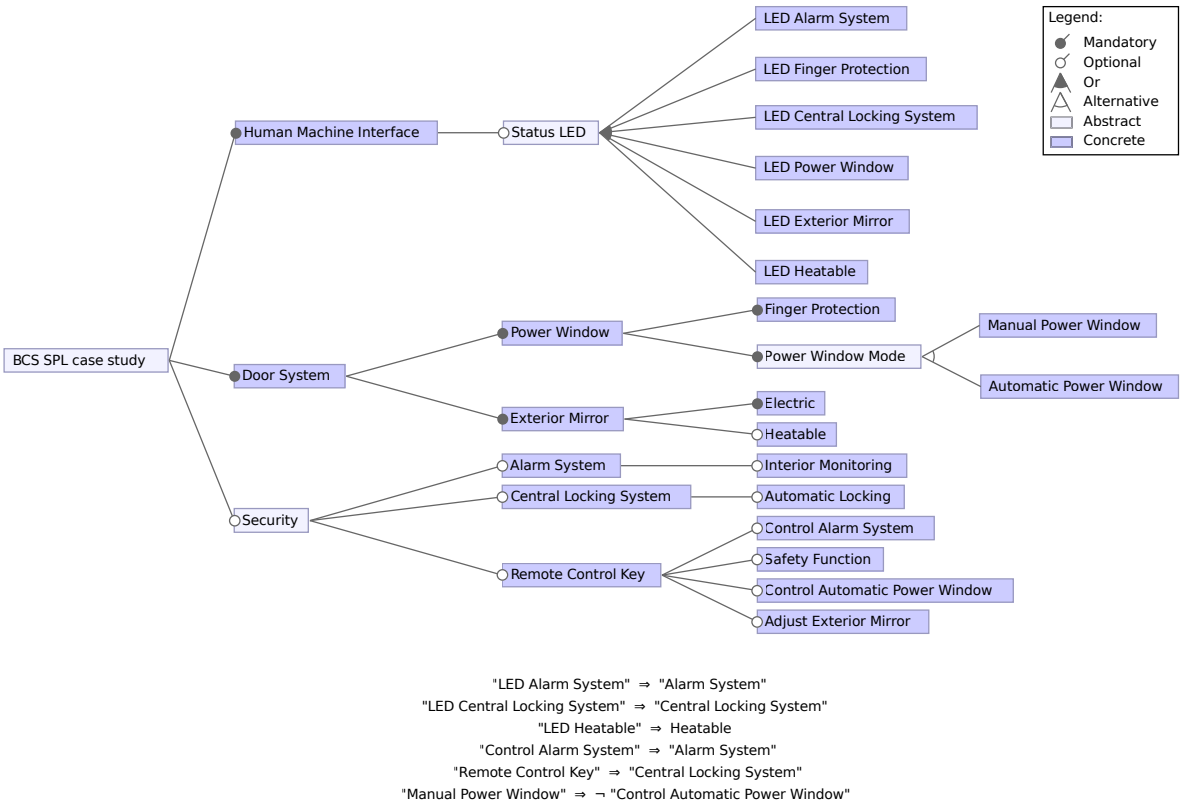


Figure 6.1.: Feature diagram of the *BCS SPL case study*

Alarm System (AS)

The *optional* `AS` feature controls the activation and deactivation of the alarm, as well as triggering the alarms. Besides, it sets the alarm silent after the alarm time elapsed. For example, the car owner does not hear the alarm and does not turn the alarm off. In this case, the alarm is set silent after a certain time.

Interior Monitoring (IM)

The *optional* `IM` feature adds functionality to the `AS` feature, which allows to monitor the interior of the car.

Central Locking System (CLS)

The *optional* `CLS` feature controls the locking and unlocking of the car.

Automatic Locking (AL)

The *optional* `AL` feature automatically locks the car, when it is driving (e.g., to prevent people from getting on your car, when you stop at a red traffic light).

Exterior Mirror (EM)

The `EM` feature provides the exterior mirror.

Electric

This feature controls the electrical movement of the exterior mirror.

Heatable (EM_heating)

The *optional* `EM_heating` feature provides heating for the `EM` feature (e.g., to deice the mirrors during winter).

Finger Protection (FP)

The FP feature checks for clamped fingers and disables and re-enables the movement of the window based on this information.

Human Machine Interface (HMI)

The HMI feature enables the interaction with the driver of the car.

LED Alarm System (LED_AS)

The *optional* LED_AS feature adds four LEDs to the HMI, in order to indicate that the AS feature is active, that an alarm was detected, that an alarm was detected and the alarm time elapsed (i.e., a alarm was set silent), or that an alarm was detected by the IM feature.

LED Central Locking System (LED_CLS)

The *optional* LED_CLS feature adds an LED to the HMI, which indicates, whether the car is locked or unlocked.

LED Exterior Mirror (LED_EM)

The *optional* LED_EM feature adds four LEDs to the HMI, in order to indicate that the EM reached its upper, lower, left-most, or right-most position.

LED Heatable (LED_EM_heating)

The *optional* LED_EM_heating feature adds an LED to the HMI, in order to indicate, whether the EM_heating feature is turned on or off.

LED Finger Protection (LED_FP)

The *optional* LED_FP feature adds an LED to the HMI, in order to indicate, whether the FP feature is active.

LED Power Window (LED_PW)

The *optional* LED_PW feature adds two LEDs to the HMI, in order to indicate the upwards and downwards movement of the window.

Power Window (PW)

The PW feature controls the movement of the window. By selecting one of the following *alternatives*, we can select the mode of the window control.

Automatic Power Window (AutoPW)

By selecting the AutoPW feature the movement of the window is started by pushing the button once. For example, when pushing the downwards button once, the window moves automatically down till it is completely open.

Manual Power Window (ManPW)

By selecting the ManPW feature the movement of the window is controlled by pressing and holding the button. For example, when pressing and holding the downwards button, the window moves downwards until it is completely open or the button is released.

Remote Control Key (RCK)

The *optional* RCK feature provides the possibility to remotely lock and unlock the car.

Adjust Exterior Mirror

This *optional* feature adds a personalization feature to the RCK, which stores the position

of the exterior mirrors for each RCK and adjusts them to these values, when the car is unlocked with the corresponding RCK.

Control Alarm System (CAS)
The *optional* CAS feature adds functionality to control the AS with the RCK (e.g., to turn a triggered alarm off).

Control Automatic Power Window (CAP)
The *optional* CAP feature adds a possibility to control the AutoPW feature with the RCK (e.g., remotely open and close the windows).

Safety Function (SF)
The *optional* SF feature adds a timer, which locks the doors again if the car was unintentionally unlocked by the RCK signal (e.g., when the RCK is carried in a trouser pocket and is activated unintentionally).

6.1.2. 150% Model of the BCS SPL Case Study

In Figure 6.2, we present the 150% architecture for the *BCS SPL case study* from [11], which we do not explain in further detail, since we are only interested in the state charts of the *BCS SPL case study* for our evaluation. These state charts represent the internal behavior of the components in this architecture and are also configured by selecting and deselecting the features in Figure 6.1 [11]. During the analysis of the component's internal state charts, this architecture helps to understand the dependencies between the components and especially the connections between them. These connections have a high impact on the internal behavior of the components (i.e., the state charts), since they emit the events, which trigger the changes in the internal behavior of the components (i.e., they trigger state changes in the state charts). For example, the emitted signals `heating_on` and `heating_off` from component EM to component LED_EMH (i.e., the component, which controls the LED for the exterior mirror heating) trigger if the corresponding LED is turned on or off.

For the *BCS SPL case study* exists a 150% state chart, which models the functionality of all components of the 150% architecture in Figure 6.2. The annotations in this 150% state chart indicate, which feature needs to be selected, in order to include the corresponding parts in a created product variant. In Figure 6.3, we present the root region of the 150% state chart. This region only contains the Root state and nothing else. The Root state is modeled as a parallel state and contains the internal functionality of all main components displayed in Figure 6.2.

In the following, we explain the configuration possibilities of the 150% state chart. All states, that cannot be configured and, consequently, only contain mandatory parts are not further explained and can be found in Section A.3.

Large Variation Points

As we can see in Figure 6.4, the functionality of each main architecture component is represented by a region in the parallel Root state from Figure 6.3. These regions only contain one state, which contains the corresponding functionality, and a default transition to this state. Using a parallel state to model the components' internal functionality allows to separate the executed behavior of the components from each other, since exactly one region exists for each component. Consequently, the system's overall state during the execution is defined by the states currently executed in the

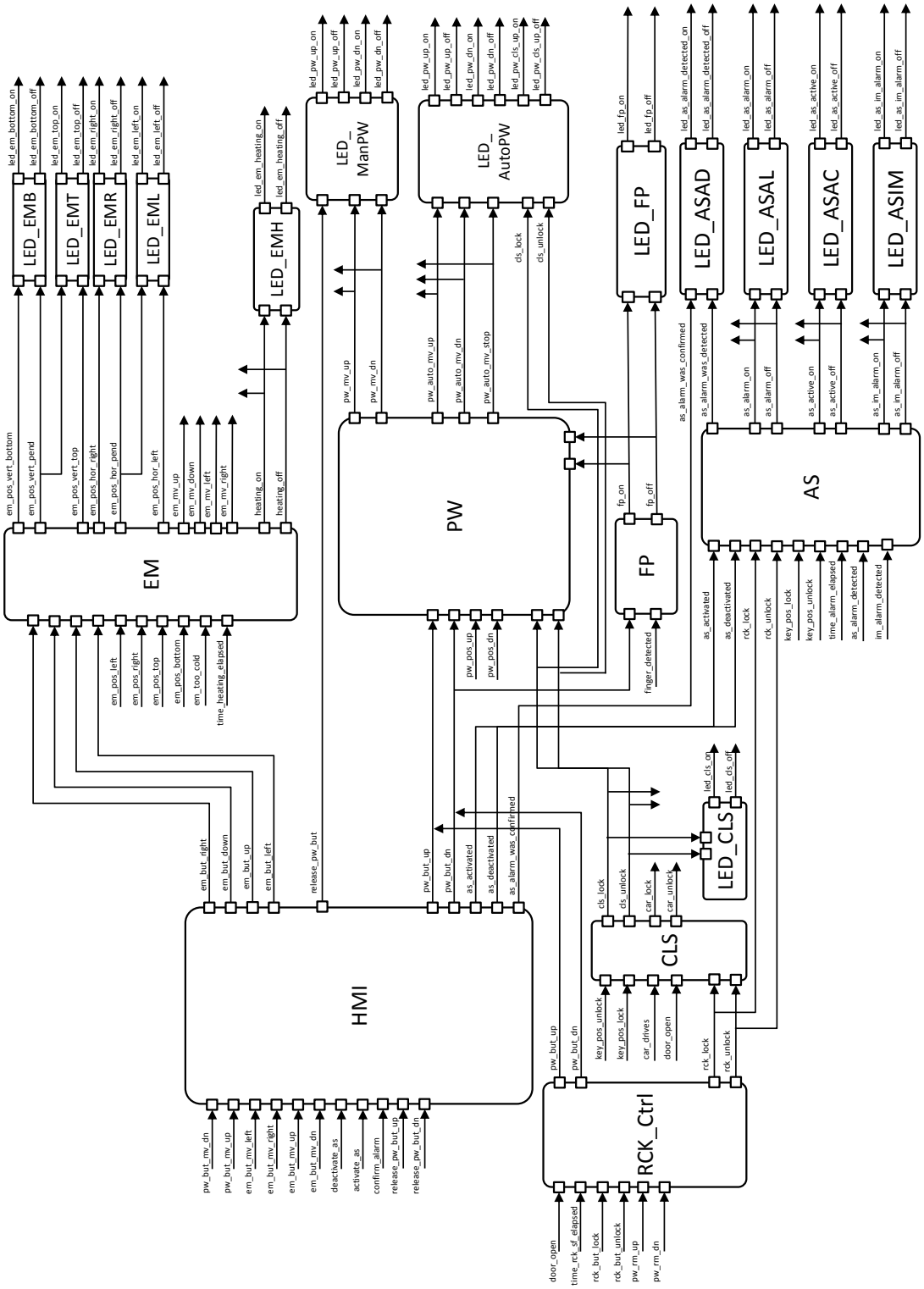


Figure 6.2.: Overview of the 150% architecture for the *BCS SPL case study*, taken from [11]

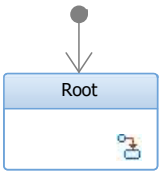


Figure 6.3.: The root region of the *BCS SPL case study* state chart

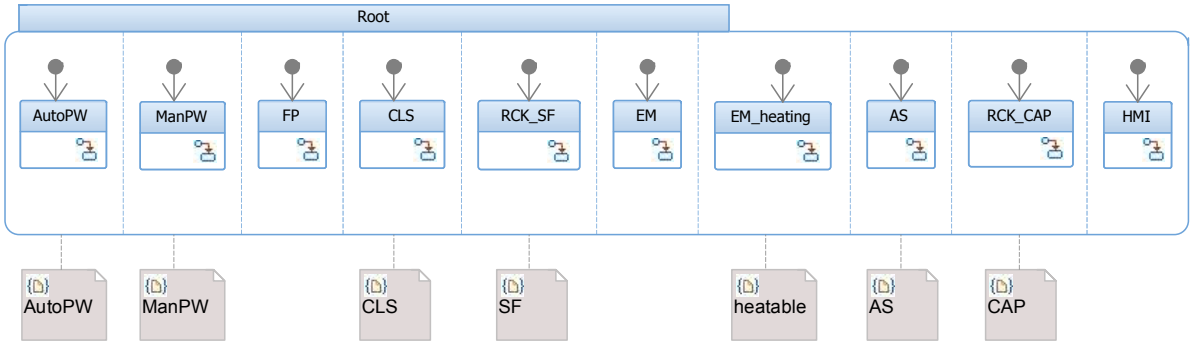


Figure 6.4.: Contents of the Root state

regions of the Root state. The labels attached to some regions define, which features select the corresponding regions for a product variant. Consequently, the AutoPW, the ManPW, the CLS, the RCK_SF, the EM_heating, the AS, and the RCK_CAP regions are only included in the state chart for a product variant, when the corresponding features were selected during generation. Besides, the cross-dependencies between these features have to be fulfilled, in order to include the corresponding regions. For example, AutoPW and ManPW are alternatives and, thus, can never be included together in the same product.

By selecting and deselecting the annotated features, we exclude complete components and their full functionality from the created state charts for the different product variants. Consequently, these *large variation points* do not allow fine granular configuration of the products, since large parts of the created state charts are affected by these configuration options. For example, deselecting the heatable feature, excludes the complete EM_heating region from the created state chart. In the following, we present all other regions, which define large variation points.

In Figure 6.5, we present the contents of the parallel HMI state. This state defines the functionality of the HMI component and can optionally include the LED state, which adds the functionality of the LEDs to display the status of the different features.

In Figure 6.6, we present the contents of the parallel LED state. The LED state adds the functionality of the different LEDs to the HMI feature. It contains annotations for the CLS feature, the heatable feature, and the AS feature.

In Figure 6.7, we present the parallel LED_PW state, which defines the functionality of the PW LEDs. The LED_PW state can be configured by selecting either the AutoPW, or the ManPW feature. Since these features are alternative to each other, the corresponding LED_PW regions are only contained according to the selected configuration.

In Figure 6.8, we show the contents of the parallel LED_AS state, which adds the functionality for the LEDs of the AS feature to the HMI feature. As we can see, it can be configured to contain the LED for the IM feature.

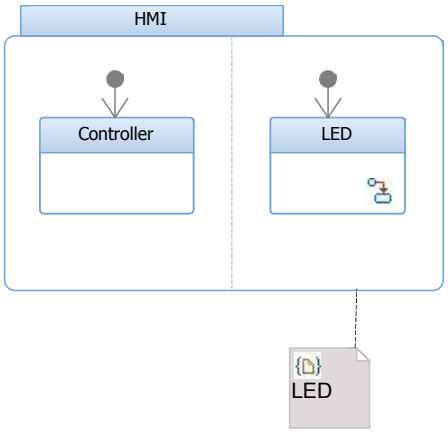


Figure 6.5.: Contents of the HMI state

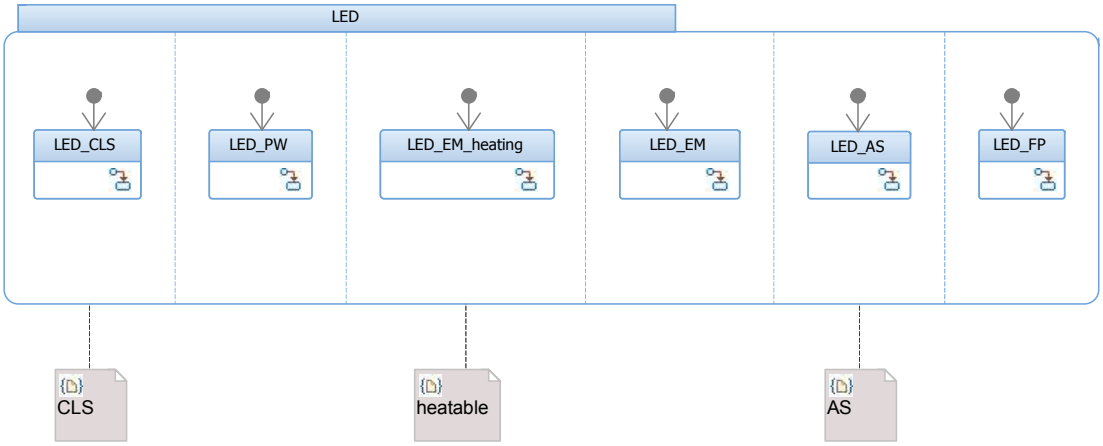


Figure 6.6.: Contents of the LED state

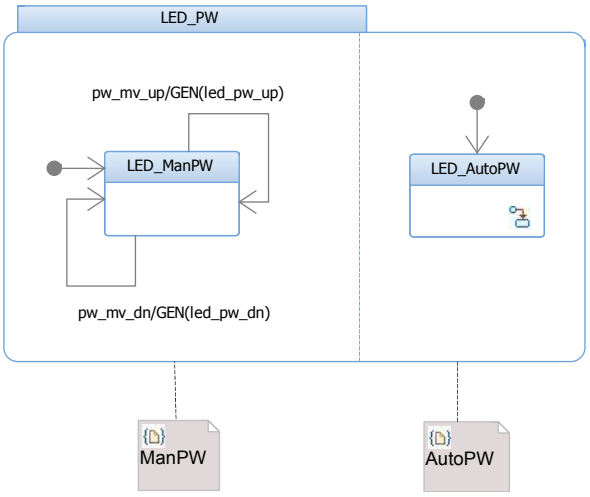


Figure 6.7.: Contents of the LED_PW state

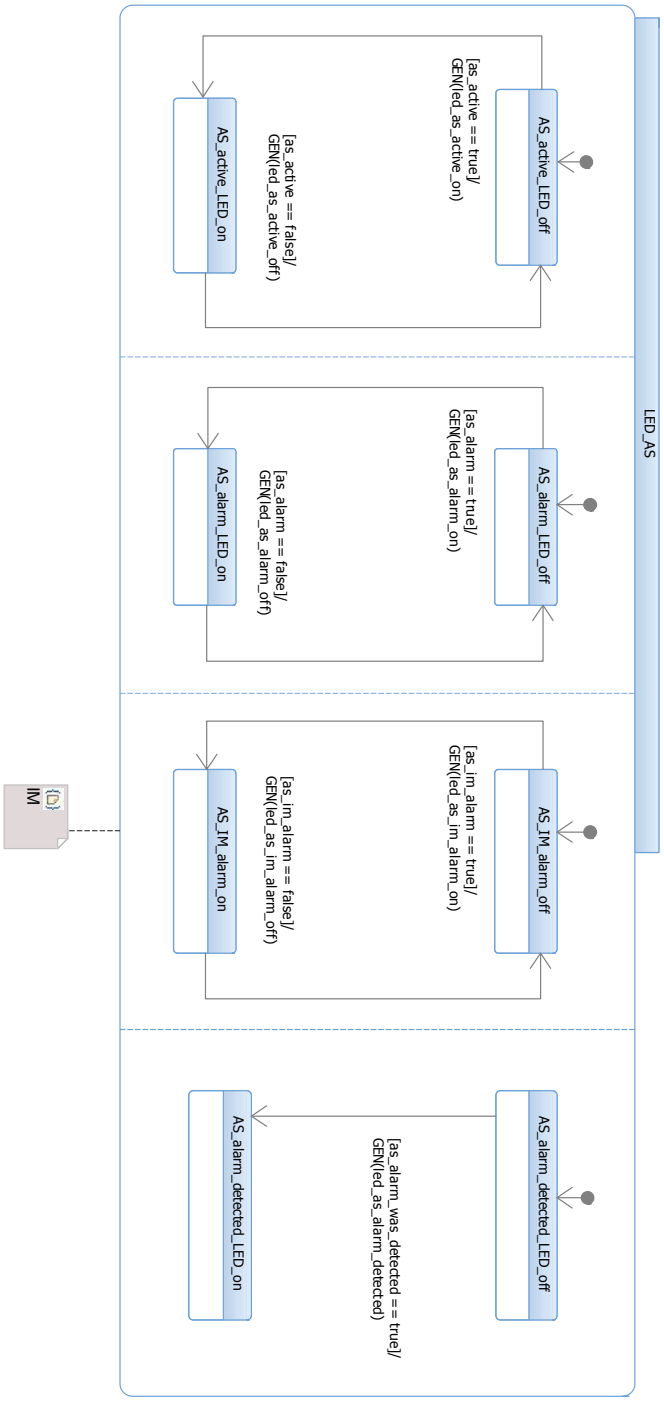


Figure 6.8.: Contents of the LED_AS state

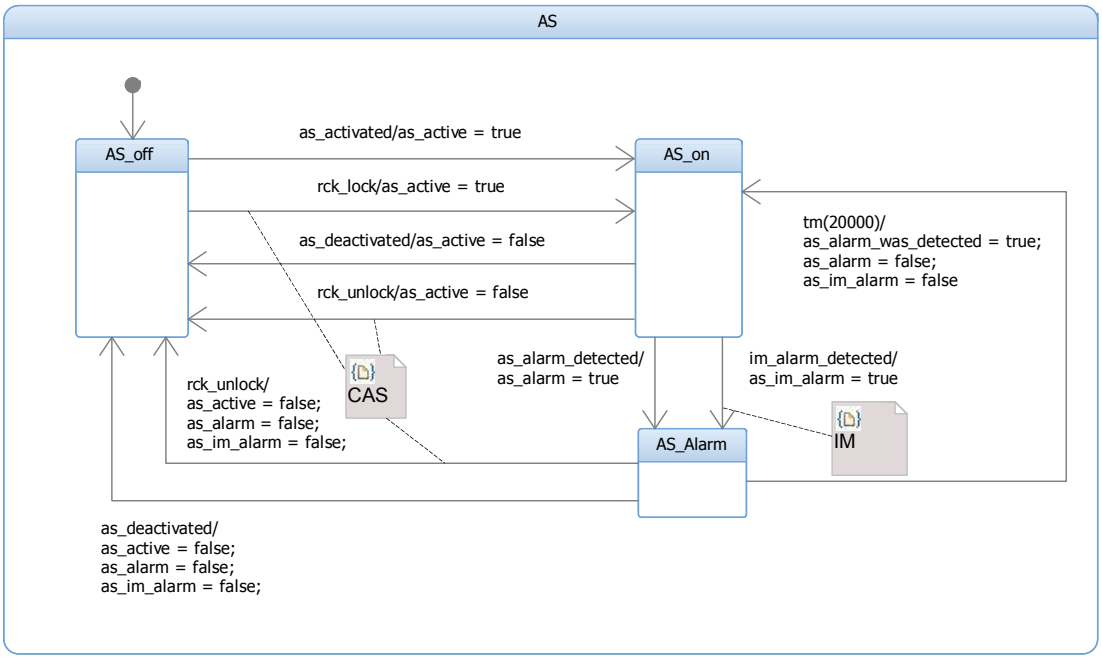


Figure 6.9.: Contents of the AS state

Fine Granular Variation Points

Fine granular variation points allow to configure products on a more fine-grained level than large variation points, since they only add or remove states or transitions when selecting or deselecting a feature. In the following we present all states, which can be configured using these fine granular variation points.

In Figure 6.9, we present the internal behavior of the AS state, which is modified when the CAS or the IM features are selected or deselected. In these cases, the correspondingly annotated transitions are added or removed from the state's contents.

In Figure 6.10, we present the internal behavior of the CLS state, which is modified when the AL, the AutoPW, or the RCK features are selected or deselected. In these cases, the correspondingly annotated transitions are added or removed from the state's contents. Furthermore, there is a mutual exclusion between some transition combinations, since two of the transitions are only added if the ManPW feature is selected, that is an alternative to the AutoPW feature, which adds a different transition. In addition there are two other transitions, which are mutual exclusive to another transition. The corresponding transitions are only added if the RCK feature is selected together with either the AutoPW feature or the ManPW feature.

In Figure 6.11, we present the internal behavior of the LED_AutoPW state, which is modified when the CLS feature is selected or deselected. In these cases, the correspondingly annotated transitions and the state are added or removed from the left region.

In Figure 6.12, we present the internal behavior of the FP state, which is modified when the AutoPW or the ManPW features are selected or deselected. As the FP system is contained in all product variants and the AutoPW and ManPW features are alternatives to each other, the FP system is always contained in the generated products and contains one of the two possible configurations (i.e., one of the accordingly annotated transitions).

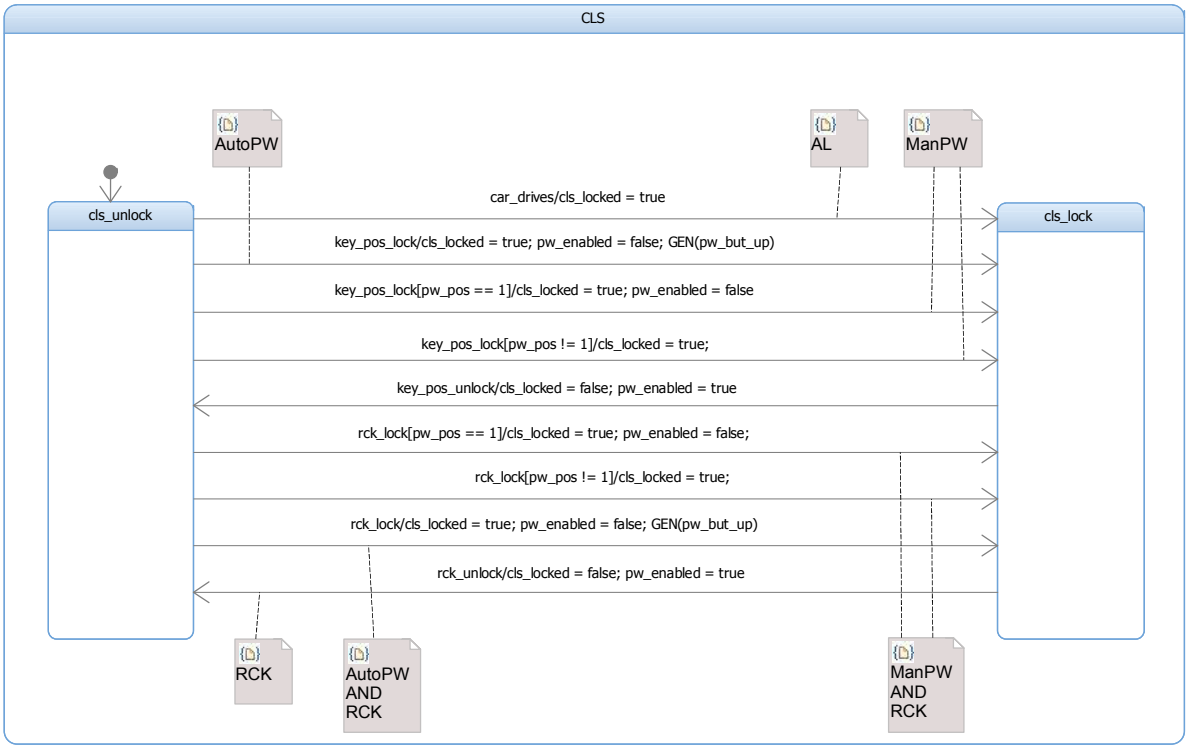


Figure 6.10.: Contents of the CLS state

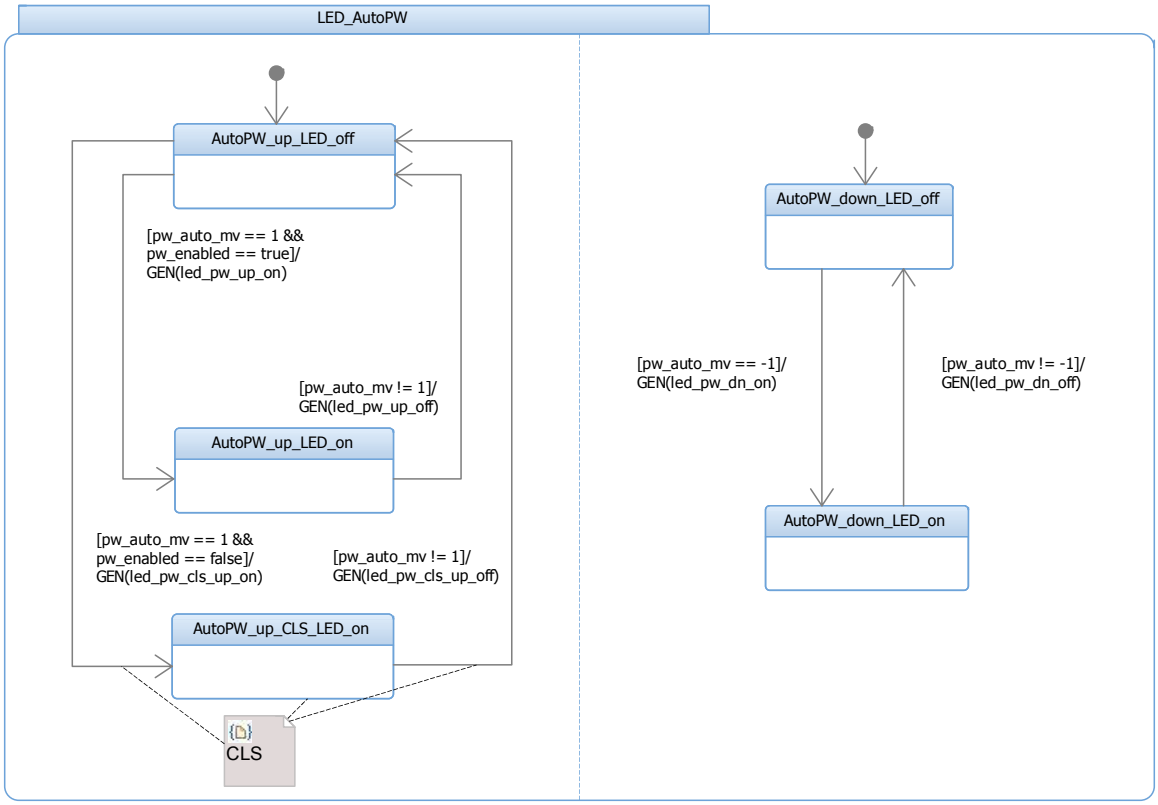


Figure 6.11.: Contents of the LED_AutoPW state

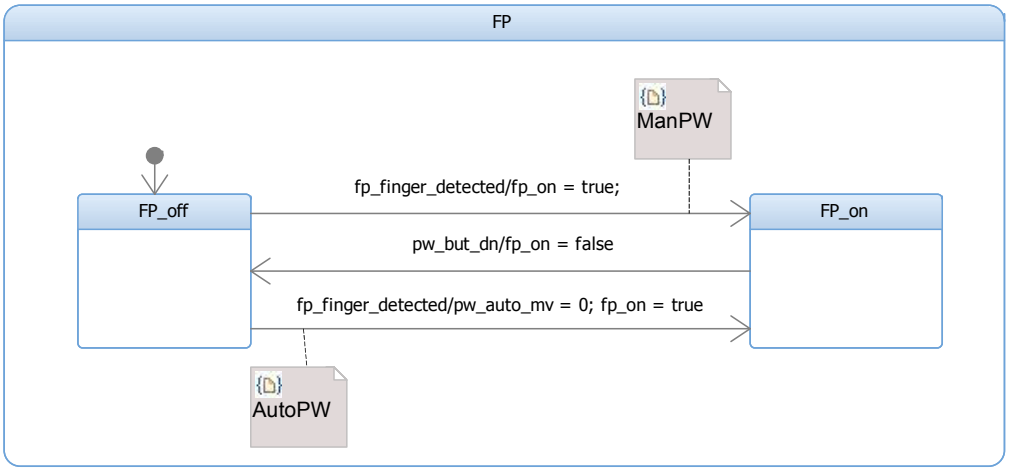


Figure 6.12.: Contents of the FP state

6.1.3. Analysis of the Variability between the State Charts

Lity et al. [11] created an overview over the product configurations of the *BCS SPL case study*. In Table A.4 in Section A.4, we present the corresponding table, which we directly took from [11]. Analyzing this table and the variability of the 150% model for the state charts (cf., Subsection 6.1.2), we created Table 6.1. This table lists all regions, which are used in the different hierarchy levels of the state charts in the *BCS SPL case study* and shows for all products, which regions are contained in the different state chart variants. The table not only represents the variable parts (i.e., *optional* and *alternative* regions) of the *BCS SPL case study*, but also the *mandatory* parts, as, for example, the `HMI` or the `EM` regions. Consequently, we get a broad overview of the state chart configurations in the *BCS SPL case study*.

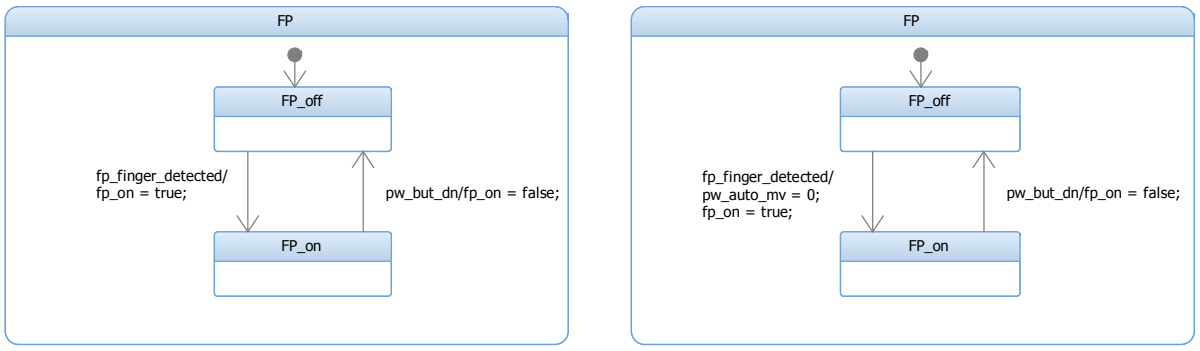
Selecting an additional feature for a product entails, that at least a new region is added to the corresponding state chart configuration in Table 6.1. For example, selecting the `AS` feature in product `P1` entails, that the `AS` region is added. Often, more than one region is added or some existing region has to be modified, because of the dependencies between different features. For example, adding the `LED_EM_heating` feature requires, that the `EM_heating` feature is included in the corresponding product. Thus, every product, which includes the `LED_EM_heating` region, also contains the `EM_heating` region. Furthermore, some features exclude each other. For example, when the `CAP` feature is added, we cannot add the `ManPW` feature, but have to add the `AutoPW` feature. Hence, all products containing the `RCK_CAP` region do not contain the `ManPW` region, but the `AutoPW` region. Important to notice for later analysis of the *BCS SPL case study* is, that product variant `P0` is the *core product*, which means, that all other products use this product as a basis and extend or modify the existing functionality.

The numbers assigned to some of the checkmarks in the table indicate, that these regions differ in some way. For example, the `FP` feature is dependent on the selected `PW` feature, because the corresponding `FP` state has to be configured according to the selected `PW`. In Figure 6.13a, we present the variant of the `FP` state, which is used for the `ManPW` feature. It differs from the variant for the `AutoPW` feature in Figure 6.13b, because the transition from `FP_off` to `FP_on` has an additional action (i.e., `pw_mv = 0`), which is only necessary if the `AutoPW` feature is selected.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
AS		✓[1]	✓[2]	✓[2]	✓[3]	✓[3]	✓[3]			✓[1]		✓[1]			✓[1]	✓[1]	✓[1]	
AutoPW		✓	✓		✓			✓	✓	✓			✓		✓	✓		
CLS		✓[4]	✓[5]	✓[6]	✓[5]		✓[7]			✓[4]		✓[8]	✓[5]	✓[8]	✓[5]	✓[5]	✓[8]	
EM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EM_heating		✓		✓		✓	✓		✓	✓			✓	✓	✓	✓	✓	✓
FP	✓[9]	✓[10]	✓[10]	✓[9]	✓[10]	✓[9]	✓[9]	✓[10]	✓[10]	✓[10]	✓[9]	✓[9]	✓[10]	✓[9]	✓[10]	✓[10]	✓[9]	✓[9]
HMI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
▶ Controller	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
▶ LED		✓		✓		✓	✓		✓	✓	✓	✓		✓			✓	
▶ LED_AS				✓		✓	✓			✓		✓					✓	
▶ AS_active_LED				✓		✓	✓			✓		✓					✓	
▶ AS_alarm_detected_LED				✓		✓	✓			✓		✓					✓	
▶ AS_alarm_LED				✓		✓	✓			✓		✓					✓	
▶ AS_IM_alarm						✓	✓			✓		✓					✓	
▶ LED_CLS				✓			✓			✓		✓		✓			✓	
▶ LED_EM				✓		✓	✓		✓	✓	✓						✓	
▶ EM_LED_bottom				✓		✓	✓		✓	✓	✓						✓	
▶ EM_LED_left				✓		✓	✓		✓	✓	✓						✓	
▶ EM_LED_right				✓		✓	✓		✓	✓	✓						✓	
▶ EM_LED_top				✓		✓	✓		✓	✓	✓						✓	
▶ LED_EM_heating				✓		✓	✓		✓	✓				✓			✓	
▶ LED_FP		✓				✓	✓		✓	✓	✓	✓		✓			✓	
▶ LED_PW				✓		✓	✓		✓	✓		✓						
▶ LED_AutoPW									✓	✓								
▶ AutoPW_down_LED									✓	✓								
▶ AutoPW_up_LED									✓[11]	✓[12]								
▶ LED_ManPW				✓		✓	✓					✓						
ManPW	✓			✓		✓	✓				✓	✓		✓			✓	✓
RCK_CAP			✓		✓					✓			✓		✓	✓		
RCK_SF			✓		✓					✓		✓	✓	✓	✓		✓	

[1, 2, 3] These are different configurations of the `AS` region.
[4, 5, 6, 7, 8] These are configurations of the `CLS` region.
[9, 10] These are different configurations of the `FP` region.
[11, 12] These are different configurations of the `LED_AutoPW_up_LED` region.

Table 6.1.: Overview over the state chart configurations of the *BCS SPL case study*



(a) The functionality of the `FP` state, when selecting the `ManPW` feature

(b) The functionality of the `FP` state, when selecting the `AutoPW` feature

Figure 6.13.: Different variants of the `FP` state

6.2. Settings used during the Evaluation

As described in Chapter 5, we need to realize a metric, implementing the `Metric` class, which provides the weights for the comparison of state chart elements, a `StringSimilarityAlgorithm`, and methods to classify the identified variability (cf., Subsection 5.6.1). Besides, we need to implement a `DecisionWizard`, which solves conflicts with ambiguous compare elements during the *Matching Phase* (cf., Subsection 5.6.3).

For the evaluation of our family mining approach with the state charts from the *BCS SPL case study*, we created a metric, which is adjusted to the settings of the *BCS SPL case study*. In Table 6.2, we present the weights, which we assigned to the different keys used in the `Metric` class. The explanations for the different keys can be found in Table A.2 in Section A.2. All values were defined using the results of our identity analysis in Section 3.5 and were adjusted to the settings of the *BCS SPL case study*. Besides, we considered the explanations in Subsection 3.5.4 to create a metric, which uses the full range to weight the elements' properties. As we can see, some of the values are set to 0.0. Either, these state chart elements are not supported by *IBM Rational Rhapsody*, which only applies to the transition type, transition priorities, and the condition action for the transition labels, or they are not used in the *BCS SPL case study*. This applies to end states, history states, and stereotypes for states and transitions.

Beside, creating the weights for the comparison, we implemented a `StringSimilarityAlgorithm`, which is used when comparing `Strings` with each other. The `EqualsSimilarity` class defines the corresponding comparison and uses the standard `equals()` method from the Java API. The implemented compare method returns 1.0 if the compared `Strings` are exactly the same, and 0.0, if they differ.

In order to solve conflicts during the *Matching Phase* of the family mining algorithm, we created an implementation for the `DecisionWizard` class. The `BCSDecisionWizard` class selects the best element from the list of conflicting elements, whose base element name is the same as the compare element name. If no such element exists, the first element in the list is returned.

After creating the compare elements during the *Comparing Phase* and matching them in the *Matching Phase*, we have to merge these elements back into a final family model. As explained in Subsec-

Key value	Weight
STATE_STATIC_NAME	0.2
STATE_STATIC_START_STATE	0.2
STATE_STATIC_END_STATE	0.0
STATE_STATIC_PARALLEL_STATE	0.1
STATE_STATIC_HIERARCHY_DISTANCE	0.25
STATE_STATIC_NEIGHBOR	0.25
STATE_STATIC_STEREOTYPE	0.0
STATE_DYNAMIC_HISTORY_STATE	0.0
STATE_DYNAMIC_DEPENDENT_ON	0.4
STATE_DYNAMIC_TRIGGERED	0.4
STATE_DYNAMIC_TRIGGERING_CHANGE	0.2
STATE_NEIGHBORHOOD_NEIGHBOR_SIMILARITY	0.5
STATE_NEIGHBORHOOD_INTERFACE_SIMILARITY	0.5
STATE_NEIGHBOR_NAME	0.2
STATE_NEIGHBOR_ACTIONS	0.8
TRANSITION_STATIC_NAME	1.0
TRANSITION_STATIC_STEREOTYPE	0.0
TRANSITION_DYNAMIC_TRANSITION_LABEL	1.0
TRANSITION_DYNAMIC_TYPE	0.0
TRANSITION_DYNAMIC_PRIORITY	0.0
TRANSITION_LABEL_EVENT	0.4
TRANSITION_LABEL_CONDITION	0.3
TRANSITION_LABEL_CONDITION_ACTION	0.0
TRANSITION_LABEL_TRANSITION_ACTION	0.3
REGION_SUBSTATE	0.5
REGION_SUBTRANSITION	0.5
STATE_STATIC	0.5
STATE_DYNAMIC	0.5
TRANSITION_STATIC	0.2
TRANSITION_DYNAMIC	0.8

Table 6.2.: Metric weights used for the evaluation of the *BCS SPL case study*

Variability	Thresholds
optional	$x = 0.0$
alternative	$0.0 > x > 0.95$
mandatory	$x \geq 0.95$

Table 6.3.: Thresholds for identifying the variability in the *BCS SPL case study*

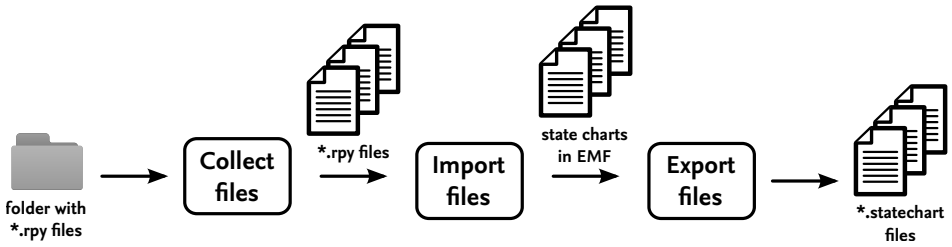


Figure 6.14.: Workflow to create `*.statechart` files from `*.rpy` files

tion 5.6.1, the `Metric` class provides methods to identify the variability from the calculated similarity. In Table 6.3, we present the thresholds, which we defined to identify the variability calculated for the compare elements in the *BCS SPL case study*, where x represents the similarity of the corresponding compare element. All compare elements with a similarity greater than 0.0 but lower than 0.95 are identified to contain *alternatives*. Compare elements with a similarity exactly equal to 0.0 are *optional* and compare elements with a similarity greater than 0.95 are *mandatory*.

As described in Section 5.7, we first have to import the `*.statechart` files for the *IBM Rational Rhapsody* files, which we want to use to evaluate our implementation of the family mining approaches. In Figure 6.14, we show the workflow to create these files. After selecting a folder, which contains `*.rpy` files, the files are imported in our internal EMF meta-model representation, and afterwards exported to the `*.statechart` files.

Afterwards, we can apply the workflow in Figure 6.15 to the `*.statechart` files generated for the *IBM Rational Rhapsody* files, which we want to consider during the evaluation of our family mining implementation for state charts. First, the `*.statechart` files are imported, then the family mining approach is applied, and finally the result is exported to a family model. Because of the reasons discussed in Section 4.5, we did not implement a family model exporter, but implemented a possibility to print the results to the console (cf., Subsection 5.6.2).

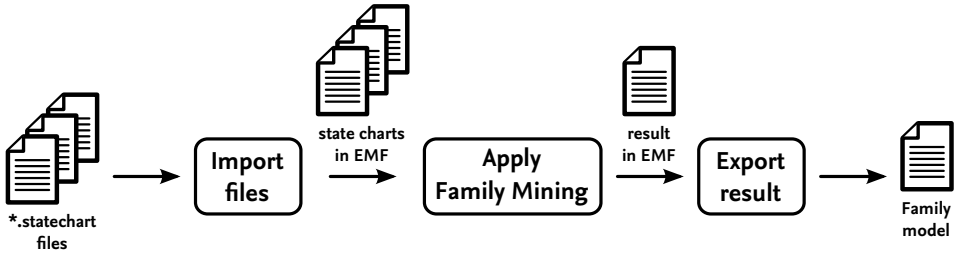


Figure 6.15.: Workflow to apply family mining to `*.statechart` files

6.3. Research Questions

In order to evaluate both approaches, the AdaptedFaMine algorithm and the ImprovedFaMine algorithm, we investigate the following *research questions (RQs)*:

Research Question 1: *Are the results, generated by the* AdaptedFaMine *algorithm, correct?*

First, we apply the AdaptedFaMine algorithm for state charts to the cases to be selected in Section 6.4. The generated results are checked for correctness, meaning we manually check, whether the identified variability between the compared state charts conforms with the human intuition.

Research Question 2: *Do both approaches, the* AdaptedFaMine *algorithm and the* ImprovedFaMine *algorithm, generate the same results? If not, where are the differences between the results and why do they exist?*

The second research question evaluates, whether the AdaptedFaMine algorithm and the ImprovedFaMine algorithm generate the same results. If the ImprovedFaMine algorithm does not create the same results, we want to identify, why the two approaches differ. For example, if there is a general problem with the ImprovedFaMine algorithm, or if the differences occur because of the changed way of processing the state charts.

Research Question 3: *How do the two family mining approaches perform compared to each other?*

The last research question evaluates, how the two family mining approaches perform compared to each other. In order to compare the performance of both approaches, we use the following measures:

Runtime in milliseconds (RT)

The runtime gives a good measure to compare the overall performance of the two family mining approaches, since creating the correct result in less time is desirable. We measure runtime for the following phases:

- Importing the `*.statechart` files
- Comparing the state charts
- Matching the created compare elements
- Merging the results back into the final 150% model files
- Overall runtime

Number of created compare elements ($\sum CE$)

This measure counts the number of created compare elements and gives a measure, how good the two compared algorithms perform, since creating the correct result with less comparisons (i.e., less compare elements are created unnecessarily) is desirable.

Number of compare elements kept after matching ($\sum CE_k$)

This measure counts the number of compare elements kept after running the matching algorithm and gives a measure how good the two compared algorithms perform, since creating compare elements, which are ruled out produces unnecessary overhead. Consequently, keeping as many of the created compare elements as possible is desirable.

Number of compare elements ruled out during matching ($\sum CE_r$)

This measure is the counterpart to the previous measure and shows, how many compare elements were ruled out during matching. It is desirable, that this number is as low as possible,

since ruling out compare elements means unnecessary overhead, because they were created unnecessarily. In order to calculate this measure, we use Equation 6.1, which subtracts the number of kept compare elements from the overall number of compare elements.

$$\sum CE_r = \sum CE - \sum CE_k \tag{6.1}$$

Number of matching algorithm calls ($\sum MAC$)

This measure counts, how often the matching algorithm is called during the execution of the family mining approach. Calling the matching algorithm during family mining more often, instead of only once at the end of the execution, means, that the number of matched compare elements during each execution is lower. Consequently, the number of possible ambiguous compare elements should be lower and, thus, the conflicts should be easier to solve.

Average number of compare elements when calling the matching algorithm ($\varnothing CE_{MAC}$)

This measure shows the average number of compare elements, which need to be matched when calling the matching algorithm. A lower number is desirable, since more compare elements, which need to be matched, are more likely to be ambiguous, because of possibly larger groups of compare elements, which have a relation to each other (i.e., they have same element from the base state chart or compare state chart). We calculate this measure by Equation 6.2. We divide the overall number of compare elements, which are processed during match algorithm calls (i.e., $\sum CE_{MAC_{start}}$) by the overall number of matching algorithm calls (i.e., $\sum MAC$). $\sum CE_{MAC_{start}}$ does not consider the compare elements, which are created during the last phase of the matching algorithm (i.e., when compare elements comparing with `null` are created for the remaining elements without matching partner) and, thus, only considers the input of the matching algorithm calls.

$$\varnothing CE_{MAC} = \frac{\sum CE_{MAC_{start}}}{\sum MAC} \tag{6.2}$$

Number of decision wizard calls ($\sum DWC$)

This measure counts the number of decision wizard calls. Calling the decision wizard indicates, that ambiguous elements were detected during matching and the algorithm could not solve this conflict by sorting the list of compare elements and matching other compare elements first. Consequently, it is desirable to keep this number as low as possible, because otherwise it might indicate a metric with badly chosen weights and thresholds.

Average number of ambiguous elements, when calling the decision wizard ($\varnothing AE_{DWC}$)

This measure shows the average number of compare elements, which are in conflict to each other when calling the decision wizard. This number should be as low as possible, since conflicts with large numbers of compare elements could be more complex to solve. In order to calculate this measure, we use Equation 6.3, which takes the overall number of ambiguous elements during the decision wizard calls (i.e., $\sum AE_{DWC_{start}}$) and divides it by the overall number of decision wizard calls (i.e., $\sum MAC$).

$$\varnothing AE_{DWC} = \frac{\sum AE_{DWC_{start}}}{\sum MAC} \tag{6.3}$$

Average number of decision wizard calls to solve conflicts ($\varnothing DWC_{SC}$)

This measure shows the average number of decision wizard calls, which are needed to solve the conflicts between the ambiguous elements. As less calls of the decision wizard indicate, that the conflict management is effective, it is desirable to keep this number as low as possible. In order to calculate this measure, we use Equation 6.4, which divides the overall number of decision wizard calls (i.e., $\sum DWC$) by the overall number of matching algorithm calls, which have to process ambiguous elements (i.e., $\sum MAC_{wAE}$).

$$\varnothing DWC_{SC} = \frac{\sum DWC}{\sum MAC_{wAE}} \tag{6.4}$$

Relation of matching calls with ambiguous elements to overall number ($\%MAC_{wAE}$)

This measure is calculated by Equation 6.5 and sets the number of matching algorithm calls *with* ambiguous elements (i.e., $\sum MAC_{wAE}$) in relation to the overall number of matching algorithm calls (i.e., $\sum MAC$). Consequently, it indicates, how well defined the metric is, because large numbers of calls with ambiguous elements mean a large number of decision wizard calls and might indicate problems regarding the used metric.

$$\%MAC_{wAE} = \frac{\sum MAC_{wAE}}{\sum MAC} \tag{6.5}$$

Relation of matching calls without ambiguous elements to overall number ($\%MAC_{woAE}$)

This measure is calculated using Equation 6.6 and is the counterpart to the previous measure. This measure sets the number of matching algorithm calls *without* ambiguous elements (i.e., $\sum MAC_{woAE}$) in relation to the overall number of matching algorithm calls (i.e., $\sum MAC$). Similar to the previous measure it indicates, how good the algorithm performs and how well defined the used metric is.

$$\%MAC_{woAE} = \frac{\sum MAC_{woAE}}{\sum MAC} \tag{6.6}$$

Relation of model sizes to runtime ($SCSrR$)

This measure is calculated by Equation 6.7 and sets the size of the two compared state charts sc_a and sc_b in relation to the runtime $RT_{a\leftrightarrow b}$, which is needed to compare them with each other. The size of a state chart $|sc|$ is defined by Equation 6.8 and is calculated by summing up the number of all elements, which need to be compared during the comparison with another state chart (i.e., all states sc_s, all transitions sc_t, and all regions sc_r). $SCSrR$ gives a good measure for the scalability of our algorithm, because we can see, how the number of compared elements correlates with the overall runtime.

$$SCSrR = \frac{|sc_a| + |sc_b|}{RT_{a\leftrightarrow b}} \tag{6.7}$$

$$|sc| = |sc_s| + |sc_t| + |sc_r| \tag{6.8}$$

In *RQ1*, we only check the results of the AdaptedFaMine algorithm for correctness, because *RQ2* checks, whether the results of the AdaptedFaMine algorithm are the same compared to the ImprovedFaMine algorithm. Consequently, we can assume, that the results of the ImprovedFaMine algorithm are correct if the AdaptedFaMine algorithm generates correct results, and the results of both approaches are the same.

Furthermore, if *RQ2* identifies, that both algorithms create the same results, the ImprovedFaMine algorithm might be an improvement to the AdaptedFaMine algorithm, given that *RQ3* shows, that it performs better compared to the AdaptedFaMine algorithm.

6.4. Selecting Cases for the Evaluation

For the evaluation of family mining for state charts, we have to select a suitable *set of cases C*, which allows us to answer our research questions. Such a set of cases consists of multiple *cases c*, which include two or more state charts, which are compared with each other by using our family mining approach. In order to properly evaluate our approach it would be ideal to consider every possible *2-tuple* (i.e., cases containing two models) from the products in the *BCS SPL case study*. As the *BCS SPL case study* contains 18 products, which we can compare with each other, we would have 153 cases, that we have to check. This number of cases c is calculated using Equation 6.9, where n is the number of products and k the number of elements, which should be contained in each tuple.

$$|C| = \frac{\binom{n}{k}}{2} \tag{6.9}$$

By using the binomial coefficient $\binom{n}{2}$ (i.e., with $k = 2$ for 2-tuples), we create the overall number of possible 2-tuple combinations (a, b), where a and b are two state charts. The binomial coefficient excludes all product combinations, where $a \equiv b$, since it only calculates the number of distinct combinations. For the comparison of two products a and b with our current family mining implementation holds $(a, b) \equiv (b, a)$, because the result of these comparisons is the same, as our algorithm always selects the state chart with the lowest number of states as the base state chart. Selecting this model as the base state chart, we try to select the state chart, which is most likely to be the core product and which, consequently, should be the basis for the comparison. Hence, we divide the calculated number of 2-tuples by two, in order to exclude all these combinations. Using Equation 6.9, we calculate:

$$|C| = \frac{\binom{18}{2}}{2} = \frac{306}{2} = 153$$

As the review of the results created by the family mining approaches is a manual task and very time consuming, we concentrate on a smaller subset of these cases. Of course, all other cases should be considered during further evaluation.

For the subset of cases, which we consider during the evaluation, we selected *27 combinations*. First, we selected all combinations, which compare the products from `P1` to `P17` with the core product `P0` (i.e., 17 combinations). Consequently, these cases should create representative results, since all products from `P1` to `P17` were created by extending or modifying `P0`. Furthermore, we selected 10 combinations, which consider the variability between the different implementation variants of the regions `AS`, `CLS`, `FP` and `AutoPW_up_LED` in Table 6.1. By checking the combinations of the different region variants, we evaluate, whether our approach for family mining of state charts correctly identifies this variability. In Table 6.4, we present all combinations with their corresponding name and

Name	Combination	$\lvert c_i \rvert$	Compared region variants			
			AS	CLS	FP	AutoPW_up_LED
c_0	P0–P1	242	–	–	(9,10)	–
c_1	P0–P2	238	–	–	(9,10)	–
c_2	P0–P3	318	–	–	–	–
c_3	P0–P4	236	–	–	(9,10)	–
c_4	P0–P5	308	–	–	–	–
c_5	P0–P6	332	–	–	–	–
c_6	P0–P7	184	–	–	(9,10)	–
c_7	P0–P8	275	–	–	(9,10)	–
c_8	P0–P9	374	–	–	(9,10)	–
c_9	P0–P10	229	–	–	–	–
c_{10}	P0–P11	296	–	–	–	–
c_{11}	P0–P12	231	–	–	(9,10)	–
c_{12}	P0–P13	258	–	–	–	–
c_{13}	P0–P14	250	–	–	(9,10)	–
c_{14}	P0–P15	238	–	–	(9,10)	–
c_{15}	P0–P16	340	–	–	–	–
c_{16}	P0–P17	193	–	–	–	–
c_{17}	P1–P3	378	(1,2)	(4,6)	(9,10)	–
c_{18}	P1–P4	296	(1,3)	(4,5)	–	–
c_{19}	P1–P11	356	–	(4,8)	(9,10)	–
c_{20}	P2–P3	374	–	(5,6)	(9,10)	–
c_{21}	P2–P6	388	(2,3)	(5,7)	(9,10)	–
c_{22}	P2–P11	352	(1,2)	(5,8)	(9,10)	–
c_{23}	P3–P6	468	(2,3)	(6,7)	–	–
c_{24}	P3–P11	432	(1,2)	(6,8)	–	–
c_{25}	P6–P11	446	(1,3)	(7,8)	–	–
c_{26}	P8–P9	467	–	–	–	(11,12)

Table 6.4.: Product combinations for the evaluation

the summed up number of state chart elements, which is compared when the corresponding case is executed (i.e., for two state charts a and b this is $|c_i| = |a| + |b|$). Besides, we show which variants of the different regions (indicated by the corresponding numbers from Table 6.1) are considered with these cases by showing the numbers annotated to the different variants in Table 6.1. All three combinations for the variants of the AS region, all ten combinations for the variants of the CLS region, and the single combinations for the regions FP and AutoPW_up_LED are considered.

6.5. Evaluation of the Results

In this section, we evaluate the results generated during the execution of the family mining for state charts with the cases discussed in Section 6.4. In Subsection 6.5.1, we evaluate the correctness of the results created by the AdaptedFaMine algorithm. In Subsection 6.5.2, we evaluate, whether the ImprovedFaMine algorithm creates the same results as the AdaptedFaMine algorithm. In Subsection 6.5.3, we evaluate the measures discussed in Section 6.3.

6.5.1. Results for Research Question 1

RQ1 challenges the correctness of the results created by the AdaptedFaMine algorithm whether they conform with the human intuition. In order to evaluate the correctness of the created results for each of the cases c_i selected in Table 6.4, we execute the following steps:

1. Execute the AdaptedFaMine algorithm to compare the state charts a and b defined by the 2-tuple in c_i.
2. Manually check the results, printed to the console, for correctness by manually comparing the state charts a and b defined by the 2-tuple in c_i and check, whether both results are the same.

In Table 6.5, we show the results r_a for the execution of the AdaptedFaMine algorithm for the 27 cases. As we can see, the execution of all 27 cases, defined in Table 6.4, terminated without any exceptions and for each of them a result r_{a_i} was returned by the implementation for the AdaptedFaMine algorithm. During the manual analysis of the printed results, we identified seven executions, whose results were not 100% correct (i.e., the results for the cases c_{17} to c_{21}, c_{24}, and c_{25}). All other results were identified to be 100% correct according to our human intuition.

6.5.2. Results for Research Question 2

RQ2 questions the correctness of the results created by the ImprovedFaMine algorithm and checks if this algorithm creates the same results as the AdaptedFaMine algorithm. This way, we can assume, that the results of the ImprovedFaMine algorithm are correct if the AdaptedFaMine algorithm generates correct results. In order to evaluate, whether both approaches create the same results for each of the cases c_j selected in Table 6.4, we execute the following steps:

1. Execute the AdaptedFaMine algorithm to compare the state charts a and b defined by the 2-tuple in c_j and store the results printed to the console in a text file *adapted*.
2. Execute the ImprovedFaMine algorithm to compare the state charts a and b defined by the 2-tuple in c_j and store the results printed to the console in a text file *improved*.
3. Use the *Compare Tool* offered by Eclipse to compare the text files *adapted* and *improved* with each other (i.e., by making a right click on them and selecting *Compare With* ▶ *Each Other*).
4. Analyze the differences identified during the comparison of the files.

In Table 6.6, we present the results r_i for the execution of the ImprovedFaMine algorithm for the 27 cases. As we can see, the execution of all 27 cases, defined in Table 6.4, terminated without any exceptions and for each of them a result r_{i_j} was returned by the implementation for the ImprovedFaMine algorithm. During the analysis of the results created by the ImprovedFaMine algorithm compared with the AdaptedFaMine algorithm, we identified, that all results are exactly the same.

Name	Combination	Terminated	Result printed	Correct result
r_{a_0}	P0–P1	✓	✓	✓
r_{a_1}	P0–P2	✓	✓	✓
r_{a_2}	P0–P3	✓	✓	✓
r_{a_3}	P0–P4	✓	✓	✓
r_{a_4}	P0–P5	✓	✓	✓
r_{a_5}	P0–P6	✓	✓	✓
r_{a_6}	P0–P7	✓	✓	✓
r_{a_7}	P0–P8	✓	✓	✓
r_{a_8}	P0–P9	✓	✓	✓
r_{a_9}	P0–P10	✓	✓	✓
$r_{a_{10}}$	P0–P11	✓	✓	✓
$r_{a_{11}}$	P0–P12	✓	✓	✓
$r_{a_{12}}$	P0–P13	✓	✓	✓
$r_{a_{13}}$	P0–P14	✓	✓	✓
$r_{a_{14}}$	P0–P15	✓	✓	✓
$r_{a_{15}}$	P0–P16	✓	✓	✓
$r_{a_{16}}$	P0–P17	✓	✓	✓
$r_{a_{17}}$	P1–P3	✓	✓	–
$r_{a_{18}}$	P1–P4	✓	✓	–
$r_{a_{19}}$	P1–P11	✓	✓	–
$r_{a_{20}}$	P2–P3	✓	✓	–
$r_{a_{21}}$	P2–P6	✓	✓	–
$r_{a_{22}}$	P2–P11	✓	✓	✓
$r_{a_{23}}$	P3–P6	✓	✓	✓
$r_{a_{24}}$	P3–P11	✓	✓	–
$r_{a_{25}}$	P6–P11	✓	✓	–
$r_{a_{26}}$	P8–P9	✓	✓	✓

Table 6.5.: Execution results r_a for the cases using the AdaptedFaMine algorithm

Name	Combination	Terminated	Result printed	Same as r_{a_i}
r_{i_0}	P0–P1	✓	✓	✓
r_{i_1}	P0–P2	✓	✓	✓
r_{i_2}	P0–P3	✓	✓	✓
r_{i_3}	P0–P4	✓	✓	✓
r_{i_4}	P0–P5	✓	✓	✓
r_{i_5}	P0–P6	✓	✓	✓
r_{i_6}	P0–P7	✓	✓	✓
r_{i_7}	P0–P8	✓	✓	✓
r_{i_8}	P0–P9	✓	✓	✓
r_{i_9}	P0–P10	✓	✓	✓
$r_{i_{10}}$	P0–P11	✓	✓	✓
$r_{i_{11}}$	P0–P12	✓	✓	✓
$r_{i_{12}}$	P0–P13	✓	✓	✓
$r_{i_{13}}$	P0–P14	✓	✓	✓
$r_{i_{14}}$	P0–P15	✓	✓	✓
$r_{i_{15}}$	P0–P16	✓	✓	✓
$r_{i_{16}}$	P0–P17	✓	✓	✓
$r_{i_{17}}$	P1–P3	✓	✓	✓
$r_{i_{18}}$	P1–P4	✓	✓	✓
$r_{i_{19}}$	P1–P11	✓	✓	✓
$r_{i_{20}}$	P2–P3	✓	✓	✓
$r_{i_{21}}$	P2–P6	✓	✓	✓
$r_{i_{22}}$	P2–P11	✓	✓	✓
$r_{i_{23}}$	P3–P6	✓	✓	✓
$r_{i_{24}}$	P3–P11	✓	✓	✓
$r_{i_{25}}$	P6–P11	✓	✓	✓
$r_{i_{26}}$	P8–P9	✓	✓	✓

Table 6.6.: Execution results r_i for the cases using the ImprovedFaMine algorithm

6.5.3. Results for Research Question 3

RQ3 compares the performance of the two algorithms (i.e., the AdaptedFaMine algorithm and the ImprovedFaMine algorithm) for each of the cases c_i selected in Table 6.4 by using the measures defined in Section 6.3.

Runtime

The first considered measure is the overall runtime, which consists of the runtimes for the *Parsing Phase*, the *Comparing Phase*, the *Matching Phase*, and the *Merging Phase*. In Figure 6.16, we present the overall runtime of the two algorithms compared to each other. As we can see, the overall runtime for the AdaptedFaMine algorithm is higher than for the ImprovedFaMine algorithm. Overall the ImprovedFaMine algorithm performs about 37.5% better for the selected cases than the AdaptedFaMine algorithm.

In Figure 6.17, we present the runtime of the *Parsing Phase* for both algorithms. The runtime varies, and we cannot definitively say if it performs better for the AdaptedFaMine algorithm or for the ImprovedFaMine algorithm.

In Figure 6.18, we show the runtime of the *Comparing Phase* for both algorithms. As we can see, the runtime of the compare algorithm for the ImprovedFaMine algorithm performs better for all selected cases than the compare algorithm for the AdaptedFaMine algorithm. Overall the compare algorithm for the ImprovedFaMine algorithm performs about 24.4% better for the selected cases than the AdaptedFaMine algorithm.

In Figure 6.19, we show the runtime of the *Matching Phase* for both algorithms. As we can see, the runtime of the matching algorithm for the ImprovedFaMine algorithm performs significantly better than the matching algorithm for the AdaptedFaMine algorithm. Overall the matching algorithm for the ImprovedFaMine algorithm performs about 83.4% better for the selected cases than the AdaptedFaMine algorithm.

In Figure 6.20, we present the runtime of the *Merging Phase* for both algorithms. Similar to the *Parsing Phase*, the runtime varies, and we cannot definitively say if it performs better for the AdaptedFaMine algorithm or for the ImprovedFaMine algorithm.

In Figure 6.21, we present the runtime distribution of the four phases compared to the overall runtime. As we can see, the *Parsing Phase* and the *Merging Phase* are more or less the same for both algorithms. Only for the *Matching Phase* and the *Comparing Phase* big differences are recognizable. The *Comparing Phase* for the ImprovedFaMine algorithm needs about 24.4% less time to create the compare elements than the AdaptedFaMine algorithm. Furthermore, the *Matching Phase* for the ImprovedFaMine algorithm needs about 83.4% less time to distinctively match the compare elements than the AdaptedFaMine algorithm.

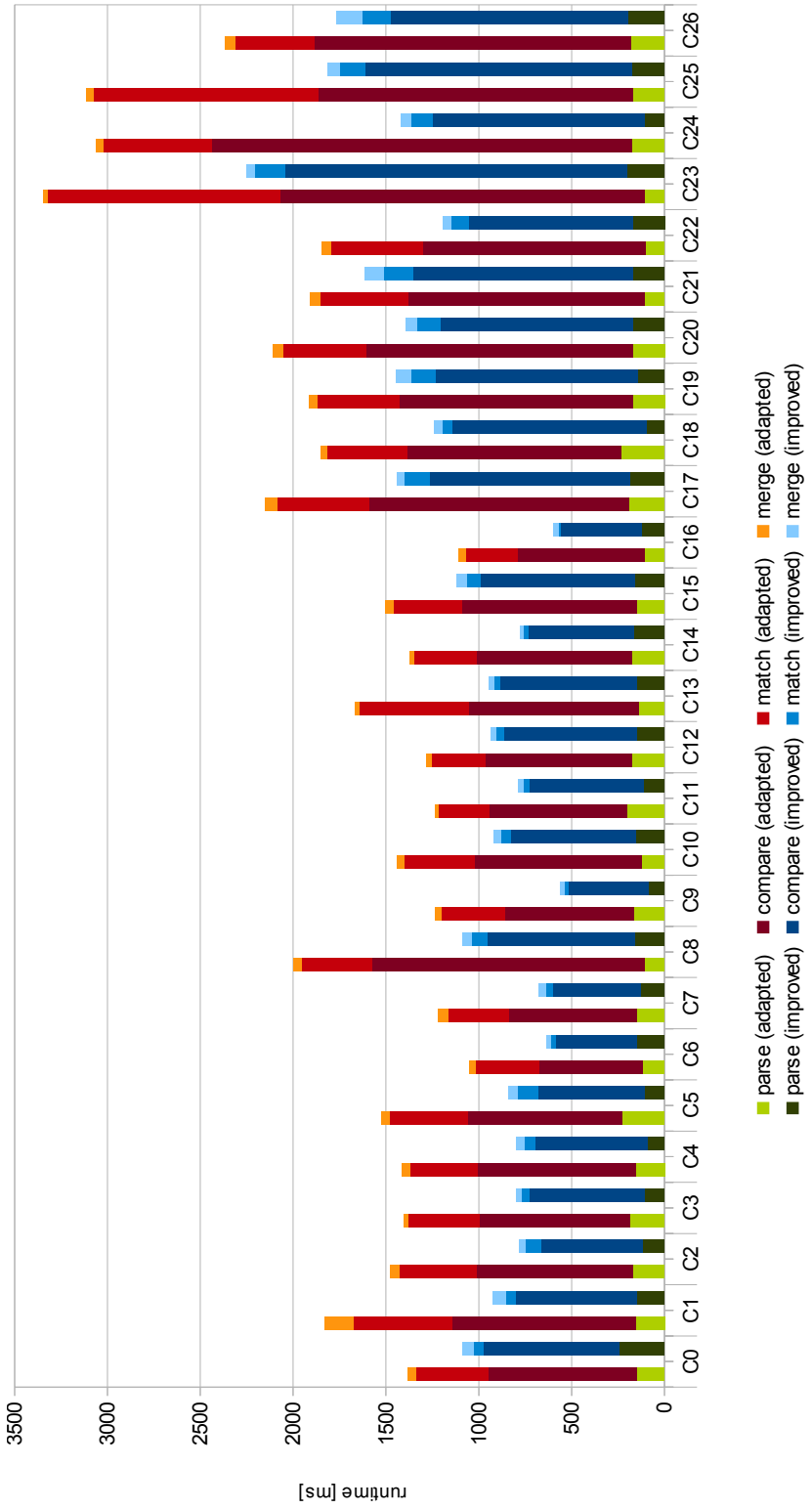


Figure 6.16.: Overall runtime

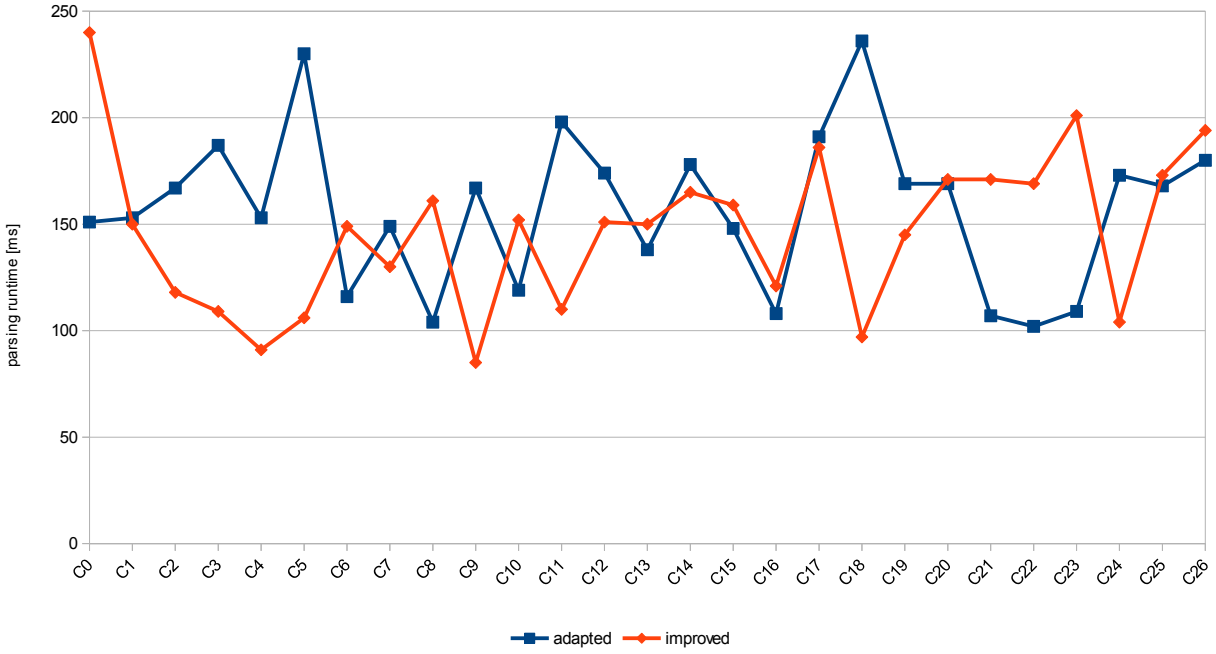


Figure 6.17.: Runtime of the parsing algorithm

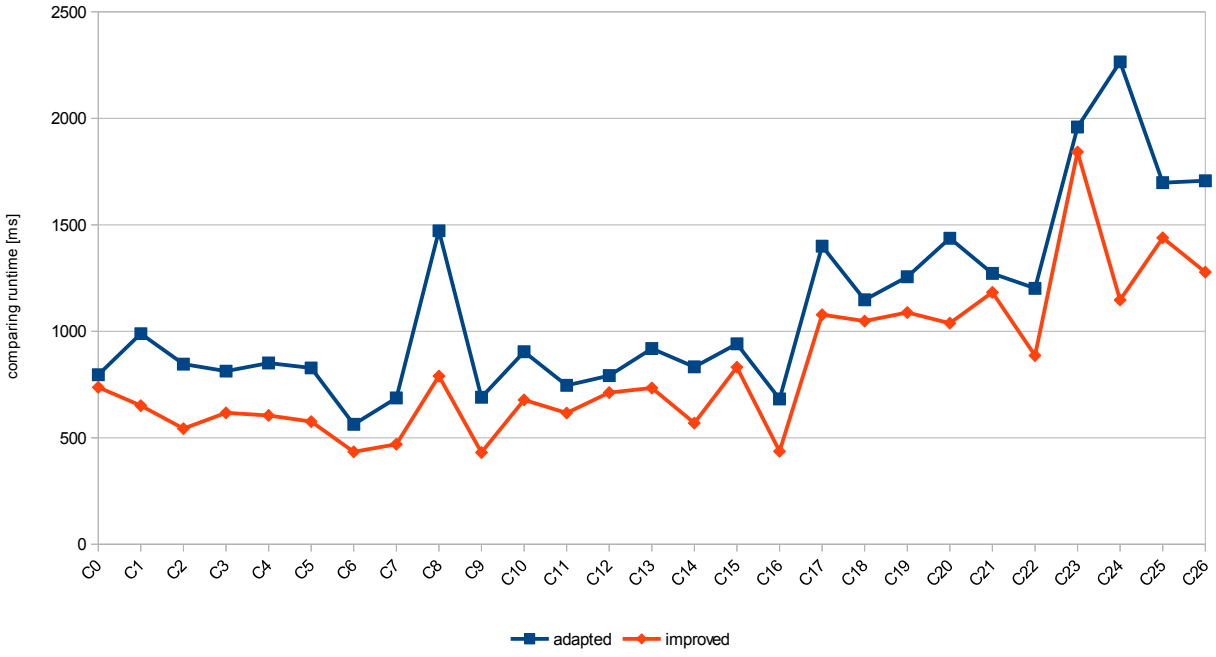


Figure 6.18.: Runtime of the comparing algorithm

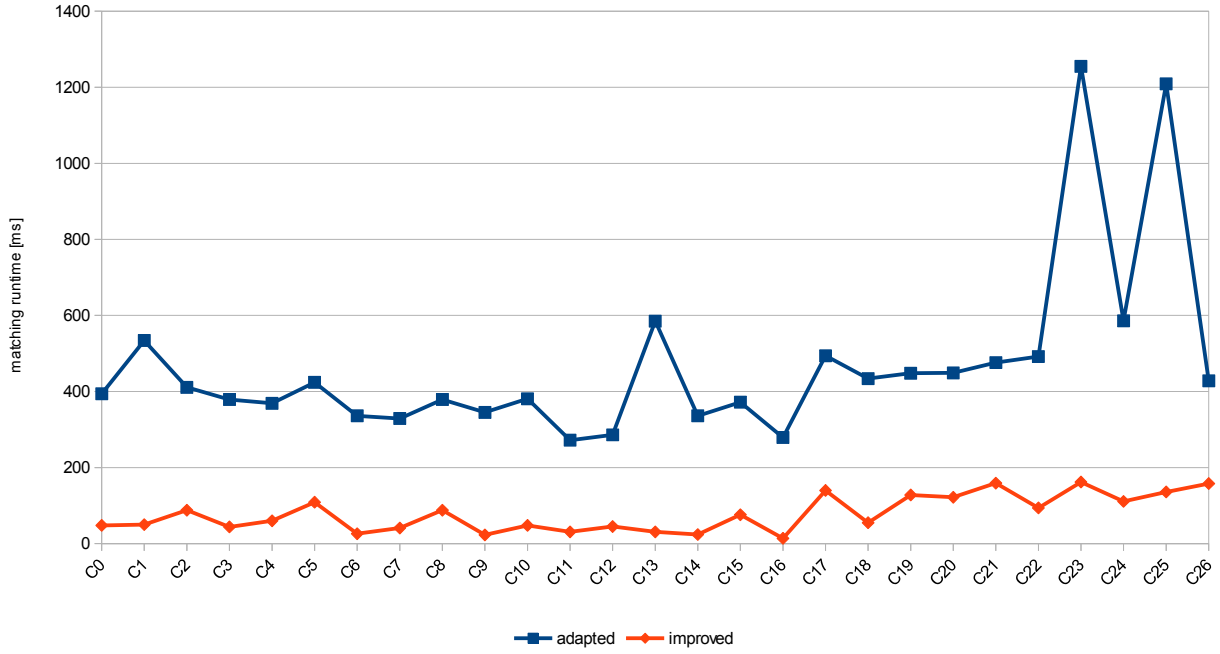


Figure 6.19.: Runtime of the matching algorithm

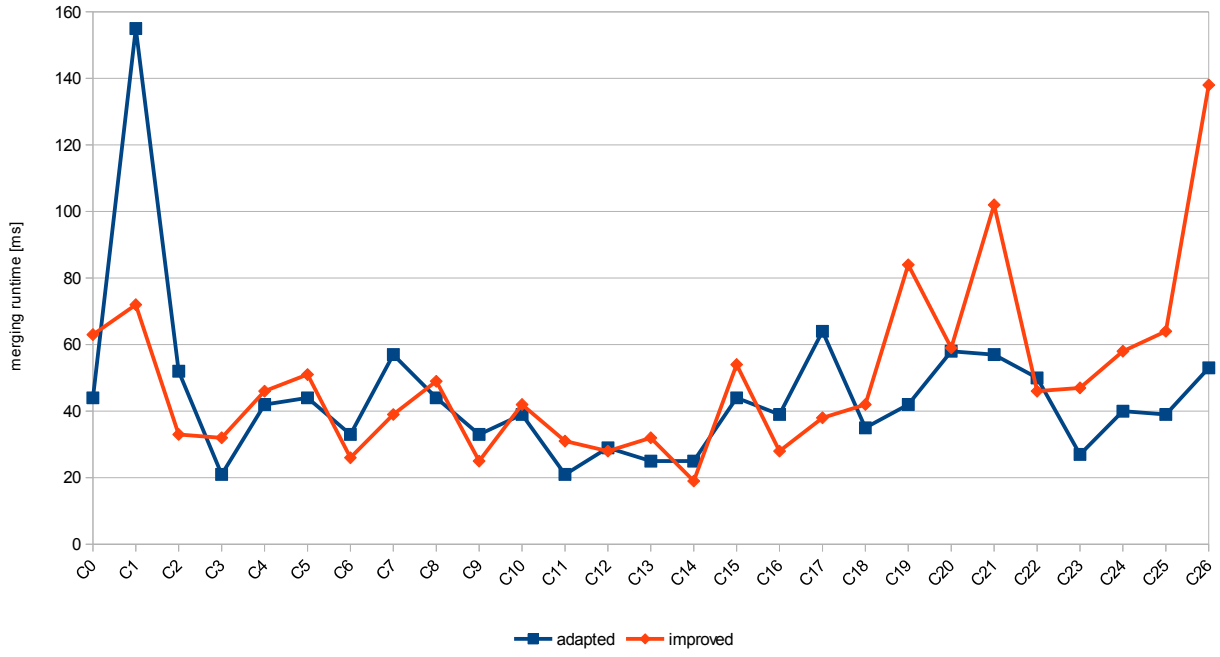


Figure 6.20.: Runtime of the merging algorithm

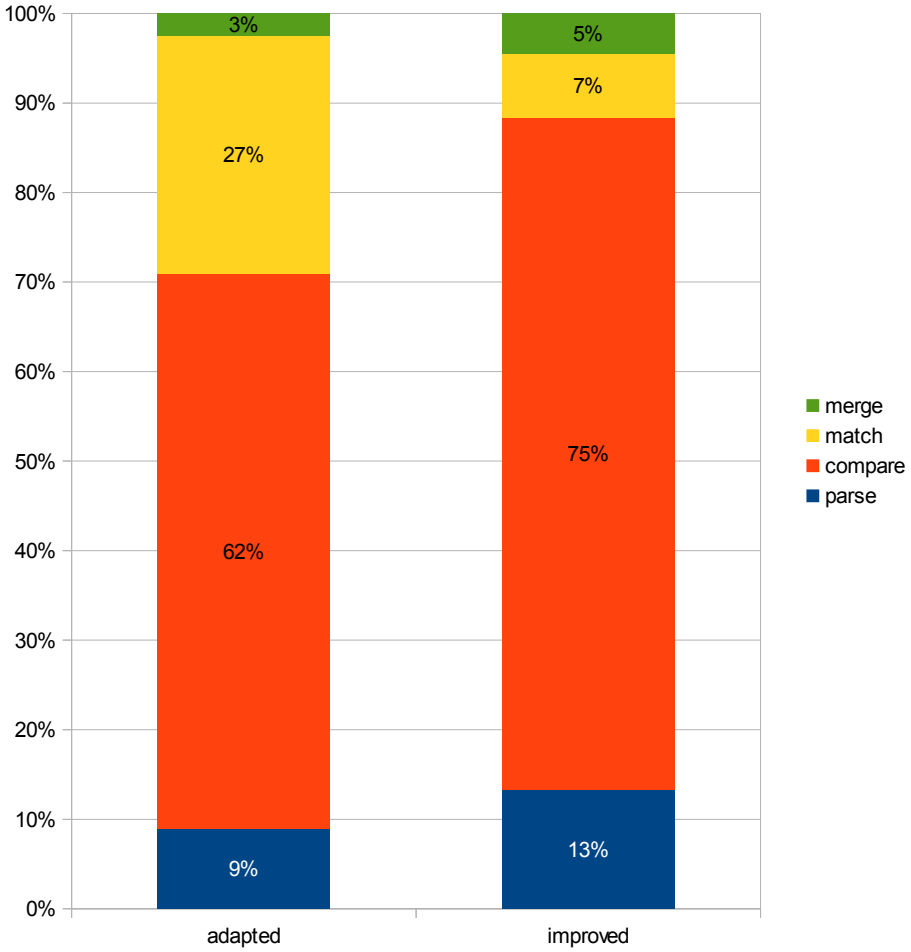


Figure 6.21.: Relation of the runtime of all phases to the overall runtime

Compare Elements

During the comparison of the state charts, compare elements are created, which store the compared elements and the corresponding similarity value calculated according to some metric. In Figure 6.22, we present the overall number of compare elements, which are created by both algorithms. The overall number of compare elements created by the ImprovedFaMine algorithm is lower than for the AdaptedFaMine algorithm. Overall the ImprovedFaMine algorithm creates about 27.5% less compare elements for the selected cases than the AdaptedFaMine algorithm.

During the execution of the *Matching Phase* some of these compare elements are ruled out, since other compare elements exist, which represent a better match. In Figure 6.23, we present the relation of ruled out and kept compare elements to the overall number of created compare elements. As we can see, the ImprovedFaMine algorithm keeps slightly more of the created compare elements than the AdaptedFaMine algorithm.

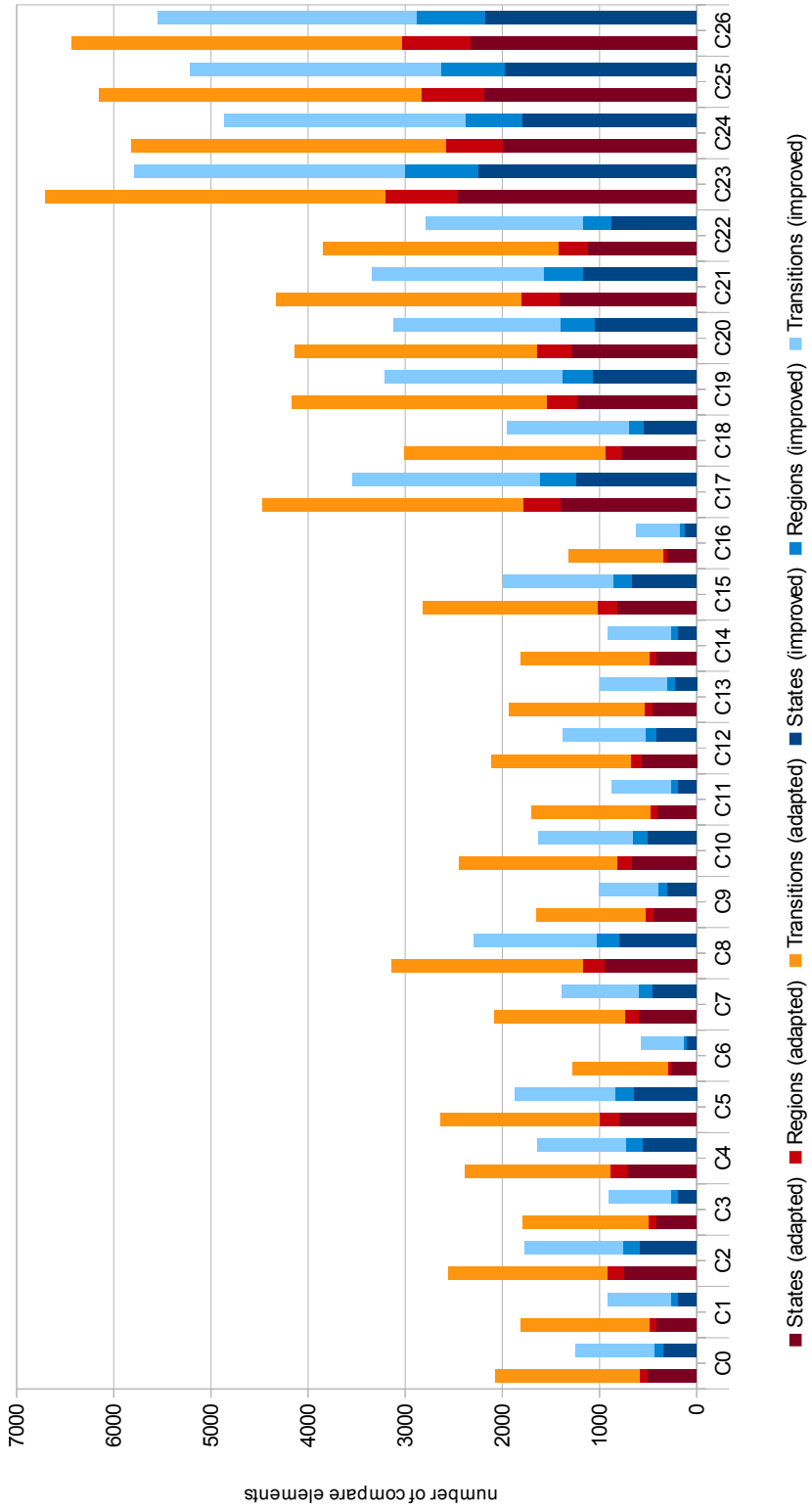


Figure 6.22.: Overall number of created compare elements

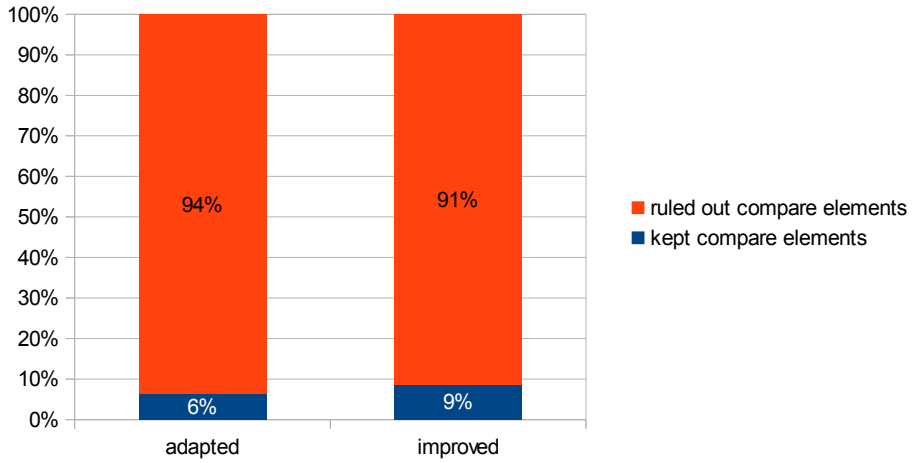


Figure 6.23.: Relation of kept and ruled out compare elements to the overall compare elements

Matching Algorithm

In order to distinctively match the created compare elements the matching algorithm is called for the corresponding algorithm. In Figure 6.24, we present the overall number of matching algorithm calls for both algorithms. As we can see, the ImprovedFaMine algorithm performs worse than the AdaptedFaMine algorithm, since it needs a strikingly increased number of matching algorithm calls to match all elements. Overall the ImprovedFaMine algorithm executes the matching algorithm about 215.8% more frequent for the selected cases than the AdaptedFaMine algorithm.

Each of these matching algorithm calls gets a certain number of compare elements, which should be matched, as an input. In Figure 6.25, we present the average number of compare elements, which are passed to the called matching algorithm for the selected cases. As we can see, the average number of compare elements, which should be processed during the matching algorithm calls for the ImprovedFaMine algorithm is significantly lower than for the AdaptedFaMine algorithm. Overall this number is about 87.3% lower for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm.

During the matching algorithm calls, sometimes no distinct match can be found for a compare element, since other relevant compare elements with the same similarity value exist. These ambiguous situations have to be solved by a decision wizard to process corresponding *ambiguous elements*. In Figure 6.26, we present for the selected cases the relation of matching algorithm calls with and without ambiguous compare elements to the overall number of matching algorithm calls. The number of matching algorithm calls with ambiguous elements for the ImprovedFaMine algorithm is significantly lower than for the AdaptedFaMine algorithm. Overall this number is about 64% lower for the ImprovedFaMine algorithm than for the AdaptedFaMine algorithm.

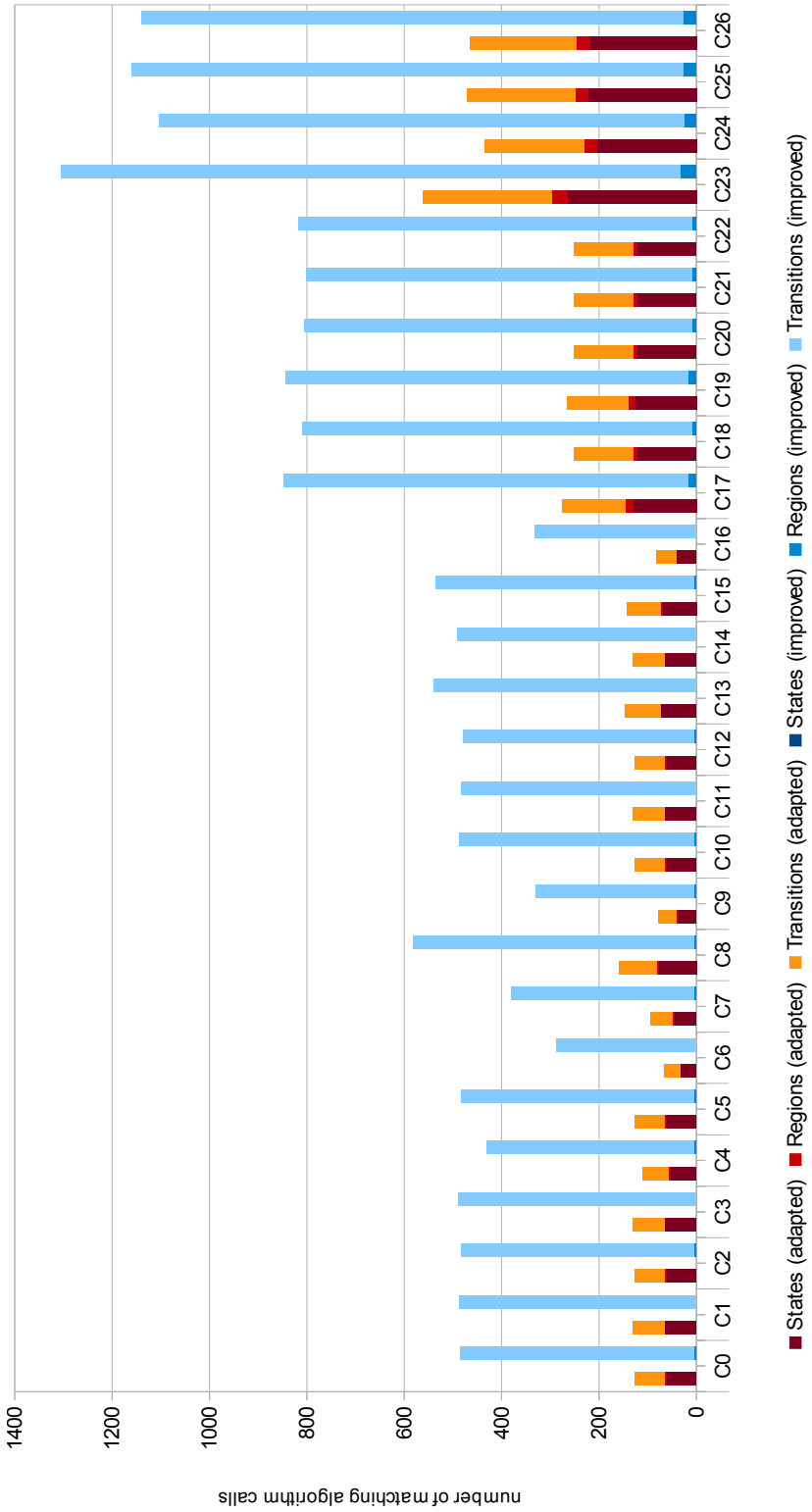


Figure 6.24.: Overall matching algorithm calls

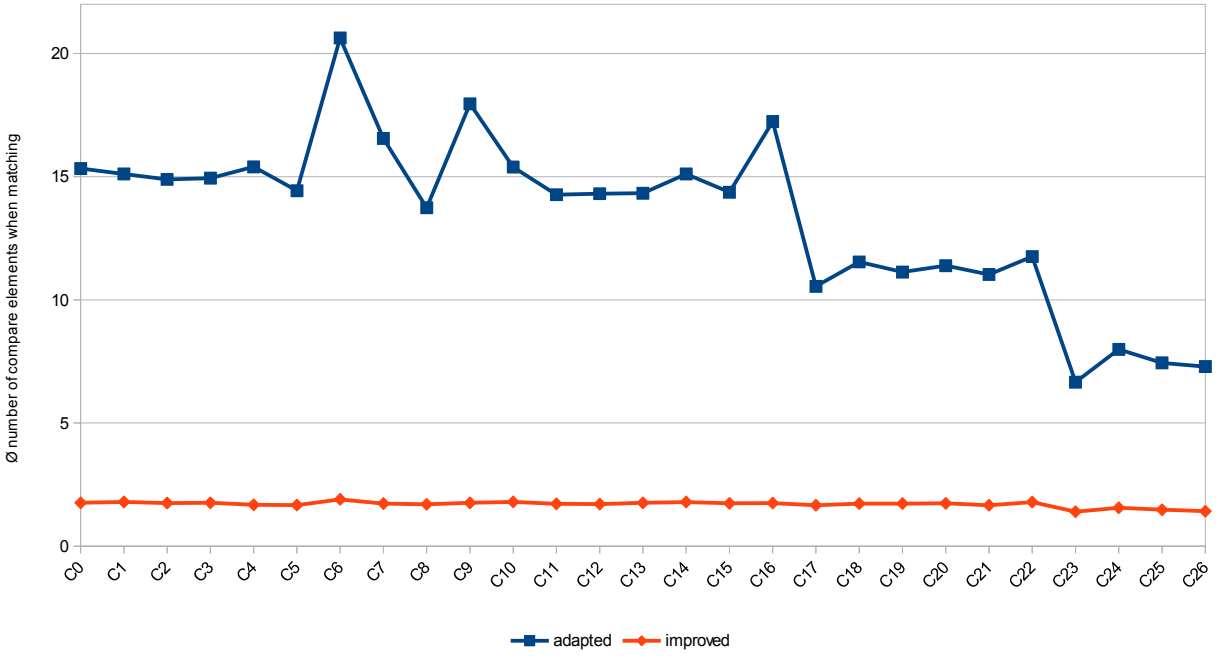


Figure 6.25.: Average number of compare elements when matching

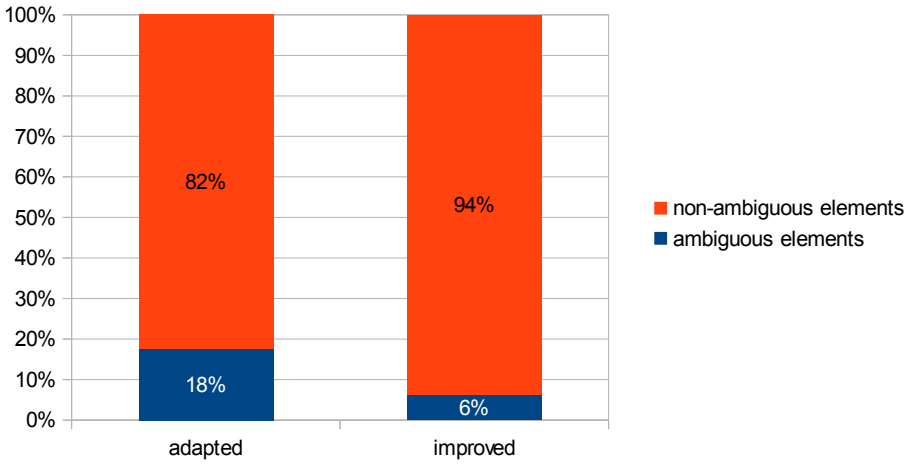


Figure 6.26.: Relation of matching algorithm calls with and without ambiguous compare elements

Decision Wizard

When ambiguous elements are identified during the execution of the *Matching Phase*, the decision wizard is called to solve these conflicts. In Figure 6.27, we present the overall number of decision wizard calls for both algorithms. The ImprovedFaMine algorithm calls the decision wizard less frequently than the AdaptedFaMine algorithm. Overall, the ImprovedFaMine algorithm executes the decision wizard about 39.5% less than the AdaptedFaMine algorithm.

Depending on the number of ambiguous elements during the decision wizard calls, the decision to select the most suitable compare element can become more complex. In Figure 6.28, we present for both algorithms the average number of ambiguous compare elements during the decision wizard calls. The ImprovedFaMine algorithm has to process a smaller number of ambiguous compare elements for the selected cases than the AdaptedFaMine algorithm. Overall, the ImprovedFaMine algorithm has to process about 85.7% fewer ambiguous compare elements during the decision wizard calls than the AdaptedFaMine algorithm.

Depending on the number of ambiguous compare elements, the decision wizard might need multiple iterations to solve all conflicts. In Figure 6.29, we present for the selected cases the average number of decision wizard calls to solve conflicts. The ImprovedFaMine algorithm needs significantly less decision wizard calls to solve the conflicts than the AdaptedFaMine algorithm. Overall, this number is about 57.6% lower than for the AdaptedFaMine algorithm.

Scalability

In order to make a statement about the scalability of our family mining algorithm for state charts, the number of compared state chart elements for each case has to be set in relation to the runtime for the comparison of the corresponding state charts. In Figure 6.30, we present a scatter plot, which shows the number of compared state chart elements for the different cases (cf. Table 6.4) in relation to the overall runtime for the execution of the family mining algorithm to compare the corresponding state charts (cf. Figure 6.21). Looking at the linear trend line for the ImprovedFaMine algorithm, we can see that about $\frac{13}{27} \approx 48.1\%$ of the markers are above this line and about $\frac{10}{27} \approx 37\%$ are below. For the AdaptedFaMine algorithm about $\frac{13}{27} \approx 48.1\%$ of the markers are above the corresponding trend line and $\frac{11}{27} \approx 40.7\%$ are below. For both algorithms, most of the markers lie in an interval of ± 300ms around the corresponding trend line and only a small subset of markers has a higher deviation (i.e., > 300ms) compared to this linear trend line.

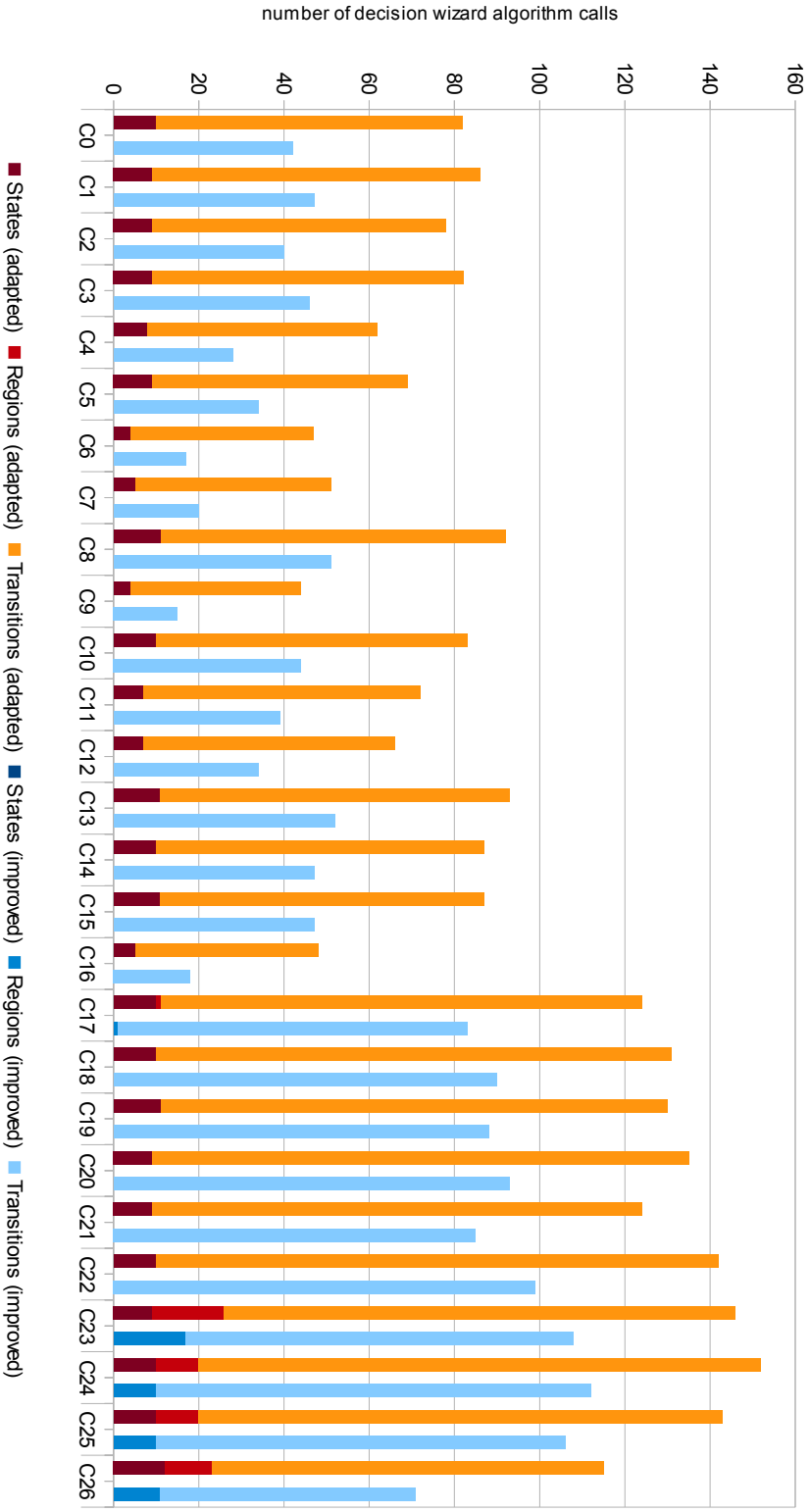


Figure 6.27.: Overall number of decision wizard calls

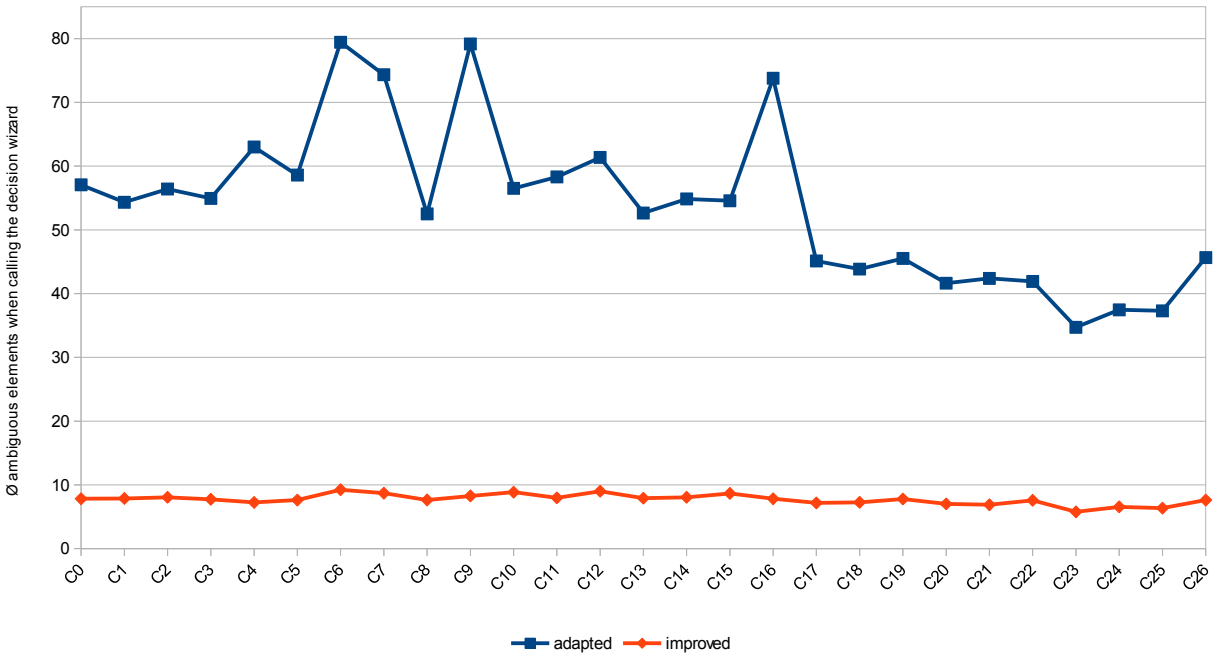


Figure 6.28.: Average number of ambiguous compare elements during the decision wizard calls

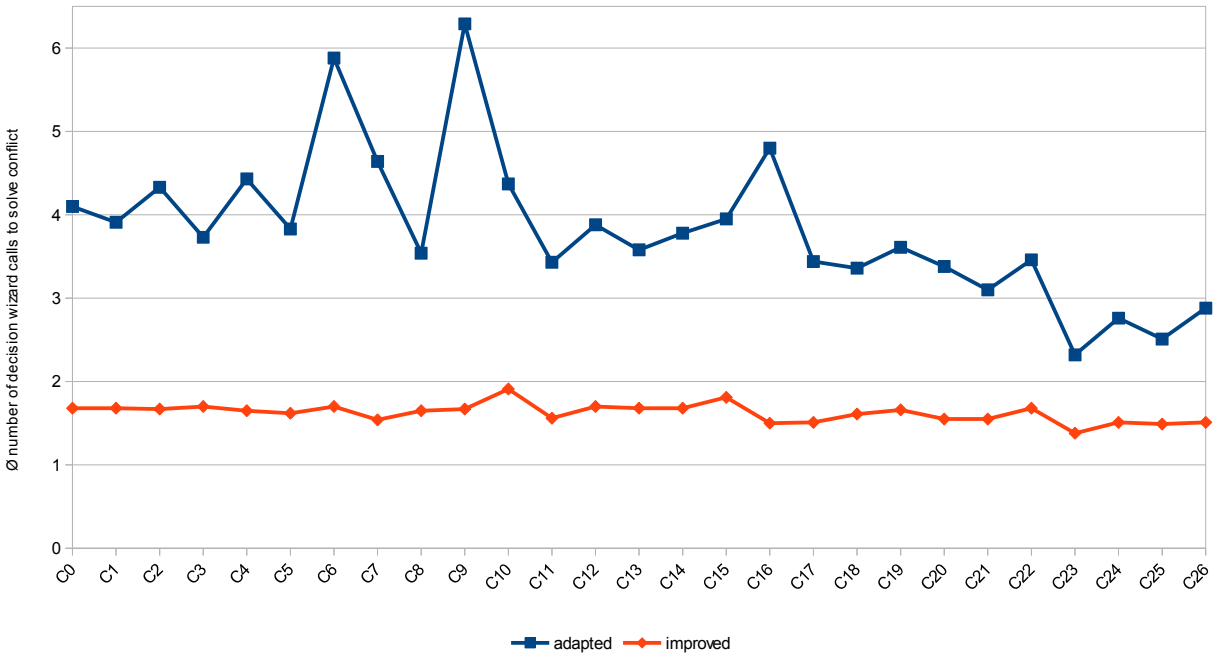


Figure 6.29.: Average number of decision wizard calls to solve conflicts

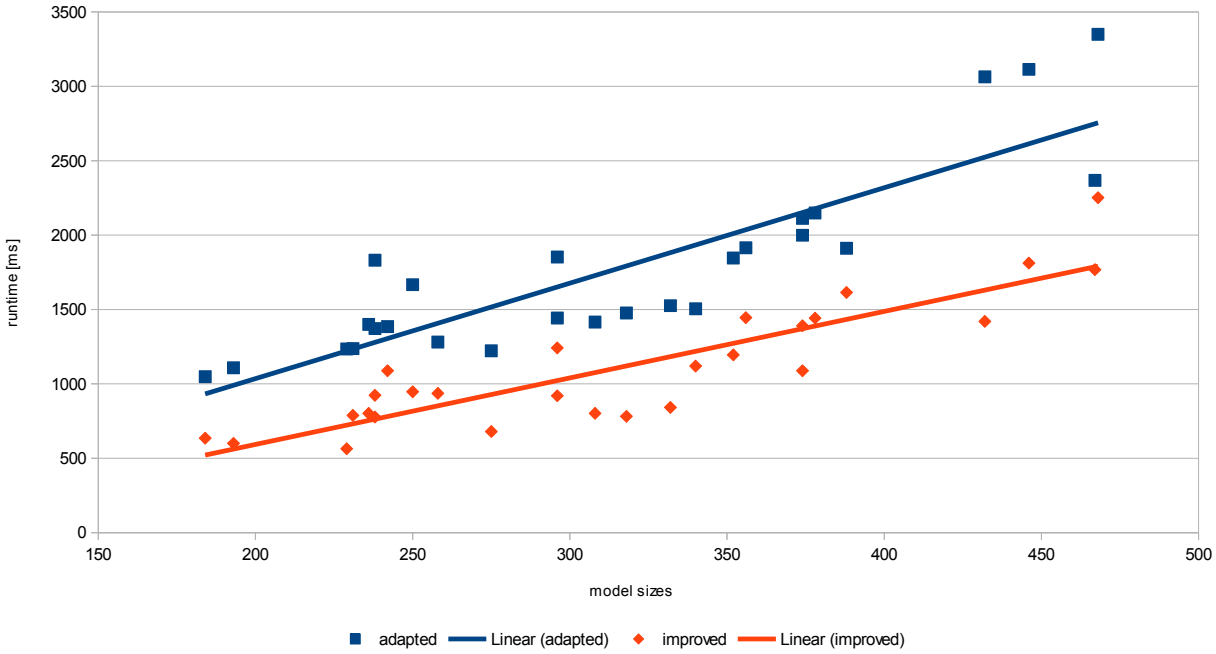


Figure 6.30.: Relation of compared state chart elements to the overall runtime

6.6. Discussion of the Results

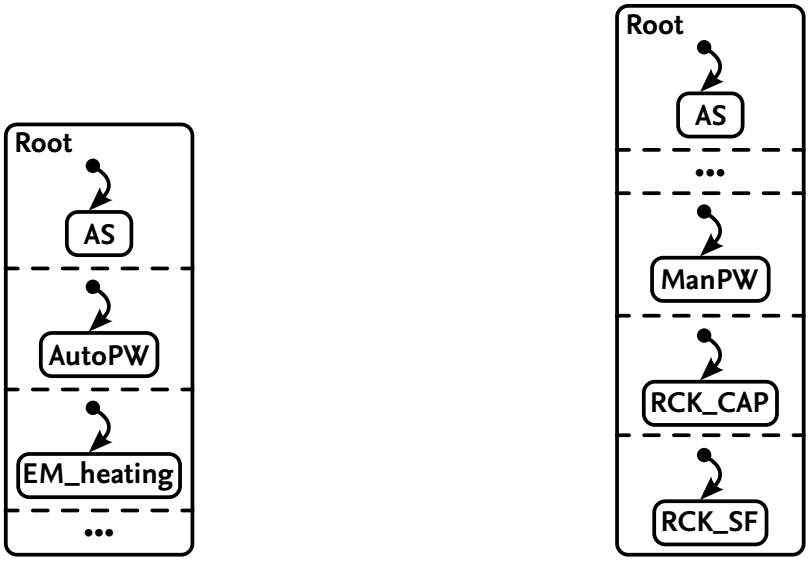
In this section, we discuss the results of the evaluation, presented in Section 6.5. In Subsection 6.6.1, we discuss the correctness of the results created by the AdaptedFaMine algorithm and the problems identified during the evaluation of these results. In Subsection 6.6.2, we discuss the correctness of the results created by the ImprovedFaMine algorithm. In Subsection 6.6.3, we discuss the gathered performance data and what we can conclude from these values.

6.6.1. Discussion of Research Question 1

As presented in Table 6.5 in Subsection 6.5.1, we identified seven cases, for which the AdaptedFaMine algorithm does not create correct results. In the following, we discuss the reasons for these incorrect results and whether they represent a general algorithmic problem or an error in our current implementation.

For the analysis of the problem, we consider an example to explain the identified causes for the incorrect results. In Figure 6.31, we present an outline of the parallel regions in the `Root` state of the product variants `P1` and `P4`, which only shows the necessary regions to explain the identified problem. Both figures show the region for the internal behavior of the `AS` component. Besides, Figure 6.31a for `P1` also shows the regions for the internal behavior of the `AutoPW` and `EM_heating` components. Figure 6.31b for `P4`, shows the regions for the internal behavior of the `ManPW`, `RCK_CAP`, and `RCK_SF` components. All other regions from both `Root` states are omitted, because similar to the `AS` region they are matched correctly and are not needed to explain the identified problem.

When executing the AdaptedFaMine algorithm for these two models, we have to compare the regions in the `Root` states. As explained in Section 4.2, we compare them by creating all possible combinations for the regions contained in the parallel state. Leaving aside the omitted regions, we



(a) Parallel regions in the Root state of P1

(b) Parallel regions in the Root state of P4

Figure 6.31.: Problem identified during the evaluation

consequently create the compare elements: (AS,AS), (AS,ManPW), (AS,RCK_CAP), (AS,RCK_SF), (AutoPW,AS), (AutoPW,ManPW), (AutoPW,RCK_CAP), (AutoPW,RCK_SF), (EM_heating,AS), (EM_heating,ManPW), (EM_heating,RCK_CAP), and (EM_heating,RCK_SF)

During the matching of the created compare elements, we identify (AS,AS) as a distinct match, because the implementation of the corresponding functionality of the regions is related, although it is *not* 100% the same (cf. Table 6.1). The same handling applies for the omitted regions, since these pairs are similarly easy to match. The next matched compare element is (AutoPW,ManPW), which is identified to be an alternative compare element, since the implementations of these two regions are similar, because they implement the same functionality but in different ways (i.e., automatic PW versus manual PW). So far all created results are correct and represent the human intuition.

Next the human intuition would expect, that EM_heating, RCK_CAP, and RCK_SF are regarded as optional elements, since they represent the internal functionality for different components of the compared systems. The matching algorithm is executed until every element has a distinct match to an element in the other model or no element is left for matching and the element has to be matched with null. Consequently, in our example the compare element (EM_heating,RCK_SF) is matched, because the basic structure of the corresponding implementation is very similar, although they implement different functionality (cf. Figure A.1 and Figure A.3). Both implementations use two states for their functionality (i.e., heating_off and heating_on versus SF_off and SF_on), which differentiates them from RCK_CAP, because this element only uses one state (i.e., CAP) for its functionality. Finally, RCK_CAP is compared with null, because it is the only remaining element.

One possible solution could be to change the lower bound of the thresholds, which define when a compare element is identified as alternative. For example, we could define, that compare elements are only regarded as alternative, when their calculated similarity value is below 95% and higher or equal to 50%. Consequently, all compare elements with a similarity value below 50% would be regarded as optional. Compare elements, which contain two compared elements and which are regarded as optional, have to be processed in a different way than the optional compare elements,

which we currently use to compare an element with `null`. For these "new" optional compare elements, we have to merge both elements to the final 150% state chart and not only one, which is not `null`. Although the discussed solution is easy to realize, we think, that it is not suitable to tackle the identified problem. From our point of view, these changes to the thresholds can evaluate compare elements wrongly, since we might identify compare elements as optional, which actually are alternative to each other. For example, when modifying the thresholds, we might correctly identify the compare element `(EM_heating,RCK_SF)`, from our example in Figure 6.31, as optional. But depending on the chosen threshold, we might additionally identify the compare element `(AutoPW,ManPW)` as optional, although it is an alternative. Thus, we would trade one wrong result for another and would not solve this issue satisfactorily.

Another solution, and also the solution, which we prefer, is to present the debatable compare elements (i.e., compare elements below a certain threshold) to the user and involve expert knowledge to classify them correctly. Although the primary goal is to automatically apply family mining to state charts, we argue, that in this case it is more desirable to involve user interaction and to create correct results than minimizing the user interaction. Of course, with this solution the number of user interactions might be higher for large models. Depending on the chosen threshold, we might present compare elements to the user, which are clearly alternatives. Similar to the implementation of the decision wizard, it might be possible to realize both, an automatic and a manual approach to process these elements. For example, we might identify certain patterns in some cases, which we could process with suitable algorithms to solve these particular situations. The described issue should be further investigated in future work, in order to find a suitable solution, which creates a correct result for such situations, but also minimizes the needed user interaction, since we want to realize an automatic family mining approach for state charts.

The results created by our current implementation are of course not 100% correct, as they do not capture human intuition. One could claim now, that the AdaptedFaMine algorithm is not realized correctly. As we discussed, this issue is not easy to automatically detect and to handle. Consequently, we argue that human interaction is needed to solve this issue.

6.6.2. Discussion of Research Question 2

We identified in Table 6.6, that both algorithms create the same results. Consequently, we argue, that the newly introduced ImprovedFaMine algorithm does not change the used family mining algorithms in such a way, that its mining results are influenced negatively so that they differ from the results created by the AdaptedFaMine algorithm. Hence, both algorithms create results, which capture human intuition, except for the issue discussed in Subsection 6.6.1.

Since both algorithms create the same results, the ImprovedFaMine algorithm logically entails the same problems as the AdaptedFaMine algorithm. This is based on the fact, that both algorithms compare multiple regions in the same manner. Thus, finding a solution for one of the algorithms and realizing it in an appropriate way, should also solve the problem for the other algorithm.

6.6.3. Discussion of Research Question 3

Since both algorithms create the same results, we can easily compare their performance, because we do not have to include any differences of their results into this discussion. In Subsection 6.5.3,

we evaluated the measures discussed in Section 6.3 for the 27 cases selected in Section 6.4. The larger part of these measures suggest, that the ImprovedFaMine algorithm performs better for the selected cases than the AdaptedFaMine algorithm.

Regarding the runtime of both algorithms, we can see in Figure 6.16, that for the execution of the selected cases the overall runtime of the ImprovedFaMine algorithm is at least equal compared to the AdaptedFaMine algorithm and for the larger part of the selected cases is significantly lower. This is also confirmed by the direct comparison of the average runtime for all selected cases, where the ImprovedFaMine algorithm is about 37.5% faster than the AdaptedFaMine algorithm.

When analyzing the composition of the overall runtime for both algorithms in Figure 6.21, we see that the *Parsing Phases* have more or less the same portion in the overall runtime of both algorithms. Also the alternation of the curves in Figure 6.17 suggests that their execution time for both algorithms is more or less the same. The same observations can be made for the *Merging Phase* in Figure 6.20. The identified behavior for the *Parsing Phase* is the expected result, since both algorithms use the same code for parsing and we used the same state charts during the evaluation. For the *Merging Phase*, this behavior emphasizes, that both approaches create the same results, since the same code is executed to merge the results into the final 150% state chart. Corresponding to these observations and to the values in Figure 6.21, we have to consider the *Comparing Phase* in Figure 6.18 and the *Matching Phase* in Figure 6.19 for both algorithms to identify the reasons for the differences in the overall runtime.

The differences in the runtime of the *Comparing Phase* can be explained with the way the compare elements are created for both algorithms. The AdaptedFaMine algorithm walks through the state charts and creates the compare elements for states, transitions, and regions corresponding to the explanations in Section 4.2. The matching algorithm is only called for the created region compare elements for each compared parallel or hierarchical state and twice to match all compare elements for states and transitions at the end of the execution. In contrast, the ImprovedFaMine algorithm executes the comparison and matching of transitions in one step and uses the results to create compare elements for the corresponding target states (cf. Section 4.6). Similar to the AdaptedFaMine approach, the region compare elements are created and matched for each compared parallel or hierarchical state. Consequently, the small performance advantage of about 24.4% of the *Comparing Phase* for the ImprovedFaMine algorithm compared to the *Comparing Phase* for the AdaptedFaMine algorithm can be explained with the state compare elements, which are directly created from the distinctively matched transitions, without creating overhead by unnecessarily creating elements, which are later ruled out.

The large difference of about 83.4% in the runtime of the *Matching Phase* for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm has multiple reasons. First, the ImprovedFaMine algorithm does not need to execute the matching algorithm for the created state compare elements, since it directly creates distinct compare elements from the target states in the matched transition compare elements (cf. Section 4.6). Besides, combining the *Comparing Phase* and the *Matching Phase* for transitions in the ImproveFaMine algorithm and calling the matching algorithm multiple times instead of only once, reduces the number of compare elements during each of the executions. In Figure 6.25, we present the average number of compare elements during the execution of the matching algorithm, which is about 87.3% lower for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm. This lower number of compare elements during each

execution of the matching algorithm in turn explains, why there are less ambiguous compare elements during the execution of the ImprovedFaMine algorithm, because less compare elements during the matching also reduce the likelihood of ambiguous elements. Consequently, we less often have to sort ambiguous elements to the end of compare elements, in order to try to solve the conflict by first matching other compare elements. Also the likelihood of ambiguous compare elements, whose conflict cannot be solved by first matching other compare elements is lower. Thus, the about 39.5% lower number of decision wizard calls (cf. Figure 6.27) and the about 57.6% lower average number of decision wizard calls to solve a conflict (cf. Figure 6.29) for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm also show, that less conflicts had to be solved using the decision wizard and that its execution was more efficient. Consequently, the overall execution time of the corresponding *Matching Phase* is reduced, since both conflict management approaches, the sorting of ambiguous elements to the end of the matched list and calling the decision wizard for the list, are very time consuming solutions.

Regarding the number of created compare elements, we can see in Figure 6.22, that about 27.5% less compare elements are created for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm. This is due to the algorithmic differences in the way the compare elements are created (cf. Section 4.6). The biggest difference is, that the ImprovedFaMine algorithm creates the needed state compare elements directly from the target states of the matched transition compare elements and, thus, only creates unnecessary state compare elements, when multiple regions are compared with each other. Some of the created combinations are ruled out together with their sub-state compare elements during the matching. This also explains, why the ImprovedFaMine algorithm keeps slightly more of the overall number of created compared elements (cf. Figure 6.23) compared to the AdaptedFaMine algorithm.

Regarding the number of matching algorithm calls, we can see in Figure 6.24, that the ImprovedFaMine algorithm executes the matching algorithm about 215.8% more often. This is due to the repeatedly called matching algorithm for the transition compare elements during the execution of this algorithm, which in turn reduces the average number of compare elements during each of the matching algorithm calls by about 87.3% compared to the AdaptedFaMine algorithm (cf. Figure 6.25). This reduced number of compare elements during the matching algorithm calls also explains the about 64% lower number of matching algorithm calls with ambiguous compare elements (cf. Figure 6.26), because as previously explained less compare elements during the *Matching Phase* reduce the likelihood of conflicts.

The reduced number of compare elements during the *Matching Phase* and the therewith reduced number of ambiguous compare elements, also explains why the decision wizard is called about 39.5% less for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm. Beside, this reduced number of compare elements is an explanation for the reduced average number of compare elements during the decision wizard calls, which is about 85.7% lower for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm (cf. Figure 6.28). Consequently, also the average number of decision wizard calls to solve a conflict is about 57.6% lower for the ImprovedFaMine algorithm compared to the AdaptedFaMine algorithm (cf. Figure 6.29).

In Figure 6.30, we presented the relation of the state chart elements to the runtime. We can see, that for both algorithms the markers are scattered, more or less, evenly around their corresponding linear trend line. For both approaches about 80% of the markers are above or below this line and the

greater part of all markers is in an interval of $\pm$300ms around the trend line. Only a small subset of all markers lies outside of this interval and has a very high deviation (i.e., $>$ 300ms) from the corresponding trend line. Thus, there is no such increase of the runtime with larger state charts, that markers deviate too much from the linear trend line and we argue that both algorithms follow a linear trend for the selected cases.

Hence, one could conclude, that both approaches follow a linear trend for state charts with increasing numbers of elements. However, when looking closely at the markers for the AdaptedFaMine algorithm, one could see, that it might also follow an exponential trend, since there are three markers for cases with about 430 to 470 elements (i.e., elements at the end of the trend line), which are significantly above the linear trend line. On the other hand, there is also one case with about 470 elements, which is significantly below the linear trend line and which might qualify this possibility. Overall, we argue, that both algorithms scale well for the comparison of the selected cases, since the corresponding markers follow a linear trend. Nevertheless, the scalability of both algorithms should be investigated further with larger state charts to confirm this linear trend for state charts with increasing numbers of elements, since we only compared cases with about 470 elements at most.

RQ2 showed, that the ImprovedFaMine algorithm creates the same results as the AdaptedFaMine algorithm. Consequently, we argue, that the ImprovedFaMine algorithm performs better, since it creates the same mining results in less time and with less effort (e.g., less compare elements are created for the same results and less conflicts have to be solved by the decision wizard). The only measure, where the ImprovedFaMine algorithm performs worse compared to the AdaptedFaMine algorithm, is the number of matching algorithm calls. However, the significantly higher number of matching algorithm calls by the ImprovedFaMine algorithm results in smaller sets of compare elements, which are matched. Thus, it reduces the number of conflicts, which have to be solved by a decision wizard and the overall runtime is reduced.

During the analysis of the scalability of both approaches, we identified, that both algorithms follow a linear trend for the selected cases and, thus, scale for these cases. As we only compared cases with about 470 elements at most, we argue, that we should investigate during future work whether this linear trend also continues for larger state charts with increasing numbers of elements.

6.7. Threats to Validity

For different reasons, the results evaluated and discussed in the previous sections are not universally valid. In Subsection 6.7.1, we discuss the *construct validity* and if the measures applied to investigate the research questions (cf. Section 6.3) are correctly gathered and interpreted. In Subsection 6.7.2, we discuss the *internal validity* and how the gathered measures might be influenced by factors, which we did not consider. In Subsection 6.7.3, we discuss the *external validity*, which challenges the generality of the results. In Subsection 6.7.4, we discuss the *reliability* of the results created during the evaluation of our family mining approach and to what extend these results are dependent on the metric and other settings, which we defined.

6.7.1. Construct Validity

We challenged in *RQ1* and *RQ2*, whether both family mining algorithms create correct results (i.e., if they capture the human intuition). In order to investigate these research questions, we manu-

ally evaluate the results of the AdaptedFaMine algorithm in Subsection 6.5.1. In Subsection 6.5.2, we transfer these results to the ImprovedFaMine algorithm by checking, whether both approaches create the same results. Since, the AdaptedFaMine algorithm mostly captures the human intuition and both algorithms create the same results, we argue that the results created by them are correct. Other researchers or developers might have a different intuition and, thus, might question the correctness of the created results. However, we have gained experience in family mining of variability during our research on this topic [6, 7, 24] and our work with industrial partners [6]. Consequently, the results should be at least close to the intuition of domain experts.

RQ3 questions the performance of both family mining algorithms and defines multiple measures to compare and evaluate them. We selected different measures for the five categories *Runtime*, *Compare Elements*, *Matching Algorithm*, *Decision Wizard*, and *Scalability*.

The measures for runtime were selected to evaluate the performance of the algorithms, which is a key aspect during the analysis of the scalability. It shows whether the algorithms are capable to handle large state charts and how the effort increases with larger models. The measures regarding the compare elements were gathered, since creating a correct result with a low number of ruled out compare elements means, that less unnecessary compare elements are created. These elements influence the overall performance of the algorithms and the memory workload. The measures regarding the matching algorithm calls and decision wizard calls analyze the performance of the matching algorithms with automated conflict solving. They show the impact of the number of compare elements during the matching on the number of conflicts and, thus, on the number of decision wizard calls. The relation of the number of state chart elements to the runtime, shows the scalability of the approach for large state charts.

All these measures were selected to give insights on the discussed criteria. Other measures might also support the results regarding these criteria and other researchers might identify more suitable ones. However, we argue, that all selected measures support the corresponding aspects during the performance analysis.

6.7.2. Internal Validity

All measures discussed by *RQ3* and gathered during the execution of the evaluation might be influenced by factors, which we did not consider when initially selecting these measures. Furthermore, we might have neglected criteria, which influence the five categories (i.e., *Runtime*, *Compare Elements*, *Matching Algorithm*, *Decision Wizard*, and *Scalability*) evaluated by *RQ3*. This might limit the validity of the measures and the statements concluded from them. However, we carefully analyzed the dependencies between the measures and the corresponding code, in order to eliminate possible influences and selected the set of measures by analyzing the most obvious influences on the considered categories. Thus, we argue, that the gathered information gives insights on the behavior of the implemented algorithms executed for the state charts of the *BCS SPL case study*. Evaluation during future work might identify further measures, which need to be analyzed in order to generalize the results for different types of state charts or larger models.

6.7.3. External Validity

We only evaluated our family mining implementation with the state charts contained in the *BCS SPL case study*. Consequently, we show, that our implementation supports the variability contained

in these particular state charts. Further state charts from other case studies might contain different variability, which we did not consider and might not support properly. Besides, we only focused on a small set of cases from the *BCS SPL case study* (cf. Section 6.4) and, thus, comparing other state charts from the case study, might reveal variability, which we did not consider properly or other problems regarding our algorithms.

The state charts of the *BCS SPL case study* do not contain all state chart elements discussed during the comparison of different state chart notations in Chapter 3. Consequently, our implementation works for the selected state charts from the *BCS SPL case study* with the contained elements, but might not support the variability in other elements (e.g., end states or state actions, which are not contained in the *BCS SPl case study*).

The sizes of the state charts contained in the *BCS SPL case study* are limited and the contained elements range from 91 to 283 elements and we were only able to compare from 184 to 468 elements with the created cases. Consequently, we can only have a limited prediction about the scalability of our approach for larger state charts with increasing numbers of elements. However, we showed, that the number of compared elements to the overall runtime follows a linear trend for the selected cases. Applying the family mining algorithms to larger models from other case studies might allow us to better understand and predict the scalability of the algorithms.

The metrics, the variability thresholds, and the decision wizard used during the evaluation of our approach are highly adjusted to the *BCS SPL case study*. Hence, other state charts from different case studies might reveal, that they are not supported by the currently used settings. Furthermore, our current metric is adjusted to *IBM Rational Rhapsody* state charts, since the *BCS SPL case study* is realized using this tool. Applying our current metric to other state charts types (e.g., state charts, created with the tools discussed in Chapter 3) might reveal an imbalance of the metric's weights for these particular tools.

According to these discussed threats, our approach does not represent a universal approach for family mining on state chats. However, our first approach allows us to gain first insights on the family mining of state charts. In future work, the gathered information can be used to improve the approach and to further evaluate it with state charts from other case studies. During this evaluation, we should investigate, whether all state chart elements discussed in Chapter 3 are supported and whether the variability in these elements is recognized correctly. In addition, this evaluation might give insights into the scalability of our algorithms for larger state charts with increasing numbers of elements.

The *BCS SPl case study* mostly contains large variation points (i.e., different regions with functionality, which are added or removed in the different product variants). Fine granular variability (i.e., variability in the states or transitions) is only contained in four of the regions in the state charts. Hence, we cannot be sure, that our approach supports all kinds of variability on this granularity level, because we only evaluated our implementation using these large variation points and a small set of fine granular variations. As discussed our approach is not a universal solution for family mining on state chats, but gives first insights on this topic. Using the state charts from the *BCS SPL case study*, we showed, that adapting family mining for state charts is feasible and that it is basically possible to identify large variation points as well as fine granular variability. During future work, we should evaluate our approach with state charts from case studies, which provide differing variation points (i.e., large one as well as fine granular ones) in order to investigate the algorithms'

capabilities of handling different combinations and granularities of variability to further support the feasibility and correctness of our family mining approach.

Another threat to validity is, that we only focused on a subset of 2-tuple cases and did not consider any other tuples, such as 3-tuple cases (i.e., comparisons with three state charts) or even larger tuples. Consequently, we cannot estimate, how our implementation behaves for such cases and whether its results would be correct. In future work all other 2-tuple cases and other tuples with three or four state charts should be considered to further investigate the correctness of our approach.

6.7.4. Reliability

Our approach for family mining of state charts is realized using metrics to compare different related state charts with each other. Metrics use heuristic weights and settings, which are highly dependent on the human intuition and the experience of the developer. Consequently, the metrics used during the evaluation are not reliable, as other developers with a differing experience might select different weights and settings, which might create other results. Furthermore, the metrics and settings used during the evaluation are highly adjusted to the *BCS SPL case study* (cf. Section 6.2). Consequently, these settings are not reliable and universally valid. However, we argue, that the used metrics were created with the biggest possible caution and only after carefully analyzing the notations of state charts (cf. Chapter 3), identifying the parts contributing to the state charts' functionality (cf. Section 3.5), and analyzing the characteristics of the *BCS SPL case study* (cf. Section 6.1). Furthermore, we have gained experience in family mining of variability during our research on this topic [6, 7, 24] and our work with industrial partners [6]. Thus, the used metrics should at least be a good approximation and create results close to the human intuition.

6.8. Summary

In this section, we described, how we evaluated our current implementation of the AdaptedFaMine algorithm and the ImprovedFaMine algorithm using the state charts from the *BCS SPL case study*. First, we described the contents of the case study, in order to better understand the relations between its state charts. After describing the settings used during the evaluation, we selected three research questions to evaluate our implementation and compare the two presented algorithms. *RQ1* questions, whether the results created by the AdaptedFaMine algorithm are correct according to the human intuition. *RQ2* uses the results of *RQ1* to evaluate, whether the ImprovedFaMine algorithm generates the same results as the AdaptedFaMine algorithm and, thus, also produces correct results. *RQ3* evaluates, how the two algorithms perform compared to each other. Selecting 27 cases with different state chart combinations from the *BCS SPL case study*, we executed the evaluation, presented the results and discussed their meaning regarding the research questions.

Overall, we identified, that both algorithms generate the same results and that they both contain one issue regarding the identification of alternative and optional regions. Afterwards, we discussed, how this issue could be solved and argued, that for this issue further user interaction is needed. During the comparison of the two approaches, we identified, that the ImprovedFaMine algorithm overall performs better than the AdaptedFaMine algorithm, since it reduces the runtime, the number of created compare elements, and the conflicts, which have to be solved with the decision wizard during matching. Furthermore, we identified, that both algorithms scale well for the comparison of larger state charts, since they follow a linear trend for the relation of compared state chart elements

to the overall runtime. Finally, we discussed threats to validity, which might limit the results and the universal validity of our evaluation. For example, we only used the state charts from one case study to evaluate our implementation. Consequently, we only know for this particular case study, that our algorithms work and might identify, that other state charts are not supported properly.

7 Conclusion

In the following, we will give a short summary of the results of this thesis (cf. Section 7.1). In Section 7.2, we discuss approaches, which are related to our family mining approach for state charts. Furthermore, we discuss in Section 7.3 ideas, which should be addressed during future work in order to further improve and evaluate our family mining approach for state charts.

7.1. Summary

In this thesis, we successfully applied the current family mining approach for block-based models [7] to state charts, in order to identify the commonalities and differences between compared state charts. We addressed *RG1* and *RG2* by analyzing the different state chart notations to identify the used state chart elements and the differences and commonalities between them. With these results in mind, we created a common meta-model representation for all analyzed notations. Furthermore, we compared the identity of block-based model elements with the identity of state chart elements. This comparison not only helps us to better understand the differences between these two model types, but also to easier adapt the algorithms for the comparison. During this analysis, we also identified, which properties of state chart elements influence the overall functionality and have to be considered during the comparison.

We addressed *RG3* by describing our approach to adapt the different phases of our current family mining algorithm for block-based models to state charts (i.e., the AdaptedFaMine algorithm). During this adaption, we introduced new algorithms to compare transitions with each other, which were not considered for the comparison of block-based models, since they only enable the data flow between the blocks, but do not add any functionality. In addition, we created algorithms to compare regions with each other. These regions enable parallel execution of functionality in state charts and also are not part of block-based models. Beside adapting the existing algorithms for the *Comparing Phase* and the *Matching Phase*, we created new algorithms to merge the mining results during the *Merging Phase* into a final 150% state chart.

Furthermore, we discussed and realized a new compare algorithm (the ImprovedFaMine algorithm) when processing *RG4*. This algorithm takes advantage of the fact, that transitions have an own identity in state charts and contribute to their functionality. Thus, we were able to reduce the overall number of compare elements by comparing these transitions and infer the correct matching of the states by comparing and matching the transitions' target states.

After explaining our current implementation for both algorithms, we addressed *RG5* by evaluating our results using the state charts from the *BCS SPL case study* [16] and executing our algorithms on a representative subset of these models. This evaluation showed for most cases that both algorithms create correct results, which correspond to the human intuition. However, in certain cases the algorithms do not identify all variability correctly, because our fully automated family mining approach has limitations compared to the human mind. Thus, the approach does not correctly identify the variability of regions in all situations and might identify them as alternatives, as it tries to match all

elements from the base state chart to elements from the compare state chart. However, in certain situations it could be more sensible to create two optional regions, instead of regarding them as alternatives to each other.

Overall, the evaluation showed for *RG6*, that regarding runtime, number of created and kept compare elements, and conflicts during the *Matching Phase* the ImprovedFaMine algorithm performs better than the AdaptedFaMine algorithm, while creating the same mining results. Thus, the ImprovedFaMine algorithm takes advantage of the discussed state chart characteristics and compares state charts more efficiently than the AdaptedFaMine algorithm. During the evaluation, we also identified, that both algorithms scale well for the selected cases from the *BCS SPL case study*, since the relation of compared elements to the overall runtime follows for both algorithms a linear trend.

7.2. Related Work

Stephan et al. [23] survey methods and applications of different model comparison approaches and only list the approach by Nejati et al. [14] for state chart comparison methods. Nejati et al. [14] describe an approach to match state charts with each other and merge the results into a single state chart. This approach is fairly similar to ours, since it uses heuristics for static and dynamic attributes to compare state chart states with each other. However, the approach by Nejati et al. [14] follows a different goal and only merges the state charts and adds limited information about the contained variability.

Although the approach by Nejati et al. [14] automatically compares, matches, and merges state charts, it provides limited information about mandatory, alternative, or optional elements, since it does not identify their variability between the compared state charts, but represents merged states as tuples (a, b), where a is a state from the base state chart and b the corresponding matched partner from the compare state chart. Consequently, there is no information about the relation of these tuples regarding the variability (e.g., whether they represent alternative or mandatory states). In addition, this approach only annotates for transitions, which are not contained in both variants, from which state chart they were merged into the final result, but does not directly add variability information. These annotations can be mapped to optional transition elements, but cannot be used to identify possible alternative transitions.

Furthermore, the approach has certain limitations compared to ours. For example, Nejati et al. [14] have to transform parallel states in the equivalent non-parallel notation in Figure 7.1. As we can see, the parallel state in Figure 7.1a is transformed to the non-parallel representation in Figure 7.1b by using interleaving and creating intermediate states to represent the parallel execution in a quasi-parallel notation. In Figure 7.1b the states `x` and `y` from Figure 7.1a are transformed to a single state `xy`. The states `x'` and `y'` are represented accordingly. In addition the states `x'y` and `xy'` are created, which are used as intermediate states during the interleaved execution. If `a` is regarded as the "first" event during this execution, the path `xy` → `x'y` → `x'y'` is taken and for `b` as the "first" event the path `xy` → `xy'` → `x'y'` is taken.

This transformation does not scale, as r^2 path combinations are created, where r is the number of parallel regions. In addition, this transformation alters the original representation of parallel states and, thus, might distort the results created during the *Comparing Phase*. In contrast, our approach can natively process these parallel states without transformations and also supports parallel states with a larger number of regions.

(a) Parallel state

(b) Non-parallel state transformation

Figure 7.1.: Transformation for parallel states, taken from [14]

Furthermore, we argue, that the tuple representation of merged states used by Nejati et al. [14] might be applicable for comparisons between two state charts, but does not scale for comparisons of multiple state charts since the states would contain increasing numbers of elements.

In addition, the approach by Nejati et al. [14] uses a total function for the comparison of states. This approach might be less efficient compared to our family mining algorithm, since all states from the compared state charts are compared with each other, whereas our approach reduces the set of possible combinations. On the other hand, Nejati et al. [14] do not create compare elements for transitions, but directly derive their variability from the matched states.

Overall, we argue, that the approach by Nejati et al. [14] is not applicable for us, as it does not provide proper variability information and has to use transformations to process parallel states, which might distort the created results.

Rubin et al. [20] describe the formal foundation for a parameterizable and configurable framework, which allows to mine feature diagrams from a range of different model types, including state charts. Rubin et al. [20] describe a special *merge-in* operator, which merges different individual products into a single product line containing a distinct feature for each merged model. Furthermore, Rubin et al. [20] proof, that their approach is able to exactly generate the input models from the mined alternative features. The described operator does not allow to generate new product variants from the mined information, since it merges the variability between the compared models into one model and annotates the parts, which belong to the different features.

We argue, that our approach could also be used to create feature models with the same alternative features as the approach by Rubin et al. [20]. During our *Merging Phase*, we could annotate for all non-mandatory parts from which state chart they were merged into the final state chart and, thus, automatically create the same annotations for the alternative features as Rubin et al. [20]. The approach by Rubin et al. [20] on the other hand, could also be adapted to create family models by using the parts common to all features as *mandatory* elements and comparing the mined features with each other to identify the *alternative* and *optional* parts. However, Rubin et al. [20] only describe a formal foundation for their feature mining approach and do not explain all algorithms for a concrete implementation. Hence, we cannot easily adopt it for our purposes.

Beside the two presented approaches, also work by Frank et al. [3] exists, which describes how state charts can be integrated into a common behavioral model view. Therefor, they split their approach into the integration of the static models (i.e., structural parts as types and attributes), which is used to identify and solve naming conflicts between the elements, and the integration of the dynamic models (i.e., the actual behavior defined by the state charts), which merges the state charts. After solving the identified naming conflicts a so-called *state relationship graph* is created. This graph shows the common functionality of the compared state charts by using different relationship operators.

Afterwards, the created state relationship graphs are used to merge the common states into single states and merging the additional states into a final state chart.

The approach by Frank et al. [3] differs from ours, since their goal is the integration of state charts into a single view. Consequently, they do not annotate the elements in the final state chart with information about the variability between the state charts, but create a single state chart, which contains the integrated view. In contrast, we are interested in the variability of the different state chart elements and argue, that the approach by Frank et al. [3] is not applicable for us.

Sabetzadeh et al. [22] present the idea of a relationship-driven framework, that could be used to merge compared models and show an example for the comparison of state charts. Sabetzadeh et al. [22] argue, that relationships should be treated as *first-class artifacts* (i.e., as objects containing the information about the relationships) during the merging of models, which helps to present the matched elements in different ways. Users can compare a set of models and manually define these relationship artifacts for them. Consequently, this framework allows to analyze different matchings between the considered models and to select the combination, which captures the human intuition best. This combination is passed to the merging algorithm, which creates the final merged model.

We agree with Sabetzadeh et al. [22], that this approach is only applicable for small models, because manually analyzing large models is a tedious and complex task, as the user looses the overview over the compared models and correctly matching the elements might get impossible. Consequently, this approach is not applicable for us, since we want to automatically mine the variability information of large models. On the other hand, we agree with Sabetzadeh et al. [22], that such a framework is useful to present the final results of an automatic mining process to the user. Analyzing the presented results, the user might be able to better understand the relations between the models and to improve the created results by manually fixing incorrect relationships.

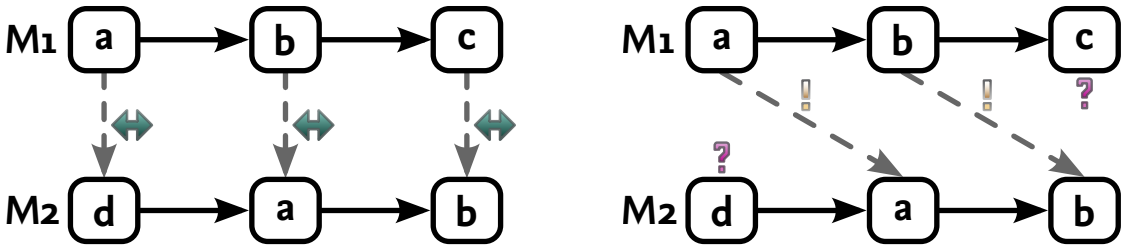
Rubin et al. [19, 21] describe an approach to compare and merge multiple models with each other. Their approach is fairly similar to ours, but they pursue a different idea. In contrast to our approach, this approach does not compare only two models at a time, but uses *n-tuples* to compare all considered n models simultaneously. Thus, at every point in the execution the whole picture of all n models is considered and not only a subset of the models (e.g., two models, as in our approach). Rubin et al. [19, 21] call this approach the *N-Way Model Merging*. The advantage of this approach is, that global results are created, which might be more accurate than the results of approaches, that compare and merge the same models step by step (e.g., our approach). Disadvantage of the approach is its runtime, since Rubin et al. [19, 21] show that their *N-Way Model Merging* approach can be reduced to *weighted set packing* and, thus is an *NP-hard* problem.

7.3. Future Work

In order to improve the results created by our family mining approach for state charts, certain issues have to be addressed. Furthermore, our family mining approach for state charts should be further evaluated, in order to improve the created results and to investigate its performance and scalability compared to other approaches.

7.3.1. Address the Identified Issue

First of all, the issue regarding the correct identification of the variability of regions (cf. Subsection 6.6.1) should be further investigated. As already discussed, the fully automatic family mining



(a) Incorrect result (b) Correct result

Figure 7.2.: Leading and trailing optional elements

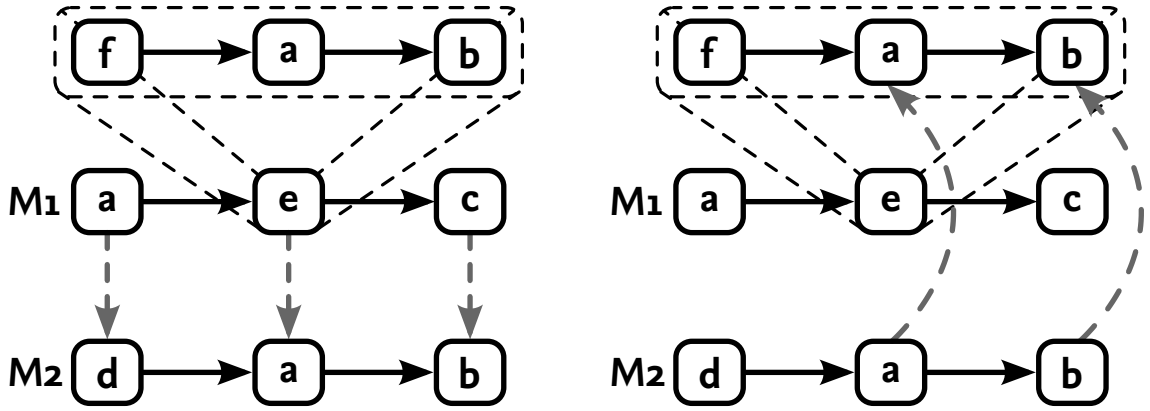
approach seems to fail for this situation. A semi-automatic solution with user interaction is most likely to create correct results. Although such a solution would break with the idea of a fully automatic family mining approach, we argue that it is more desirable to have more user interaction and correct results, rather than a fully automatic approach and incorrect results.

7.3.2. Address Special Cases in the Variability Identification

In addition, we should consider the improvements realized by Jockusch [9] for our block-based family mining algorithm. Jockusch [9] realized additional algorithms to handle certain special cases in the identification of variability. First of all, Jockusch [9] developed algorithms to detect optional leading and trailing elements for the data flow. In Figure 7.2a, we present a comparison, which shows these problematic elements. When comparing the two models `M1` and `M2`, our current family mining algorithm matches the blocks `(a,d)`, `(b,a)`, and `(c,b)` as alternatives, because it processes the data flow sequentially from the start to the end. However, the human intuition would expect the result in Figure 7.2b, which matches the blocks `(a,a)` and `(b,b)` as mandatory elements and identifies the blocks `c` and `d` as leading and trailing optional elements. By virtually splitting the models in so-called *windows* (i.e., smaller sub-models) and moving them along the data flow, the algorithms of Jockusch [9] identify these elements correctly.

Besides, Jockusch [9] is able to identify model parts, that were encapsulated during their clone-and-own phase within a subsystem and moved from their hierarchy level to another hierarchy level. In Figure 7.3a, we present an example for blocks that are moved across hierarchy levels. For this example, our current family mining algorithm matches the blocks `(a,d)`, `(e,a)`, and `(c,b)` as alternatives and does not consider the blocks in the hierarchical block `e`. The improved algorithm by Jockusch [9] uses the windows from the previous example and moves them across the hierarchies to identify such blocks. In Figure 7.3b, we present the expected result, which correctly matches the blocks `a` and `b`, that were moved across the hierarchy levels by introducing an additional subsystem.

Our current family mining algorithm also lacks support for so-called *bridges*. These bridges represent optional blocks, which were added between two other blocks. In Figure 7.4a, we present how our current family mining algorithm treats these bridges. As we can see, it correctly matches the blocks `(a,a)`, but identifies the blocks `(b,d)` as alternatives and the blocks `c` and `d` as optional elements. However, the human intuition would expect a result as in Figure 7.4b, where the blocks `(a,a)` and `(d,d)` are matched and the blocks `b` and `c` are identified as optional elements. Jockusch [9] identifies these bridges correctly by using the same window ideas as before and aggregating the gathered information for an overall image.



(a) Incorrect result (b) Correct result

Figure 7.3.: Detecting blocks across hierarchy levels

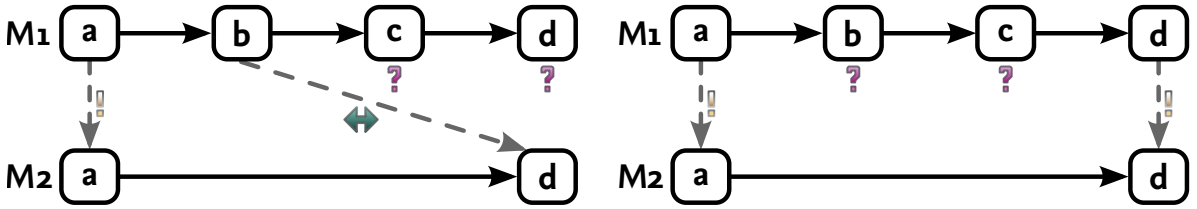
All three special cases should be considered in future work to improve the family mining of state charts, since they represent realistic modifications of state charts, which were created using clone-and-own approaches.

7.3.3. Create an Improved Family Model Representation

In addition, we should further investigate family model representations to find a suitable representation to export our family mining results. As discussed in Section 4.5, this is not a trivial task and different aspects regarding usability, readability, and accuracy have to be considered. Important is that developers can easily use the representation for their analysis (i.e., the usability should be intuitive) and understand the represented results without much effort (i.e., the results are presented in a readable way). However, the presentation of the results should not be abstracted to much, in order to be as accurate as possible, because otherwise important information might get lost. Consequently, we have to find a trade-off between an accurate and simple representation.

7.3.4. Evaluation

As discussed in Section 6.7, further evaluation should be conducted with state charts from other case studies and different settings to investigate the results created by our algorithms. During this evaluation, we might identify further issues, which should be addressed in future improvements of the algorithms. A starting point is to evaluate all 2-tuple combinations of the 18 models from the *BCS SPL case study* and to create combinations with more than two models (e.g., 3-tuples). Beside further evaluating our approach with the state charts from the *BCS SPL case study* and taking other



(a) Incorrect result (b) Correct result

Figure 7.4.: Detecting optional elements between other blocks

case studies into account, we could utilize a generator to randomly generate state charts with different sizes, which are related to each other. With these artificially generated state charts, we would be able to control certain important parameters, which influence the relation of the created state charts. Possible parameters could be the size of the compared state charts (i.e., to further evaluate the scalability of our algorithms and whether the relation of the compared elements to the overall runtime still follows a linear trend for larger models), the number of small variation points in the generated state charts (i.e., to further evaluate, whether our algorithms correctly identify all contained variability), and the degree of relation between the generated state charts (i.e., to investigate, how good the algorithms perform for state charts, which are differing a lot).

7.3.5. Compare with other Approaches

As discussed in Section 7.2, the approach by Rubin et al. [20] can be adapted to create family models and our approach can be used to mine the same features as the approach by Rubin et al. [20]. In future work, it could be interesting to identify, whether our approach for family mining of state charts is somehow related with the formal foundations by Rubin et al. [20] and whether the approaches are able to create the same family models and feature models after adapting them to the discussed ideas. Consequently, we might be able to transfer their proof to our family mining approach and show, that we are able to generate the exact input state charts from the generated feature models and family models, respectively.

7.3.6. Compare with N-Way Model Merging

A special case in related work is the *N-Way Model Merging* approach by Rubin et al. [19, 21]. As discussed in Section 7.2, this approach has certain advantages and disadvantages compared to our approach, as it creates the mined results by comparing all models at the same time and not only two models as our family mining approach. During future work, it could be interesting to compare our approach with the *N-Way Model Merging* approach by Rubin et al. [19, 21], in order to identify, how these approaches perform compared to each other. For example, the correctness of the results should be compared, since taking all models into account, when comparing them, might produce better results than only comparing two models at a time. Furthermore, the runtime is an important measure, since the approach by Rubin et al. [19, 21] is *NP-hard*. Thus, depending on the number of compared models and their size, this approach might fail to create a result, or might at least be slower than our approach.

7.3.7. Adapt Ideas from other Approaches

In Section 4.3, we discussed that our approach might not be able to directly match compare elements during the *Matching Phase* because of ambiguous elements. In these cases, our approach needs assistance by a decision wizard to solve these conflicts. We argue, that the framework by Sabetzadeh et al. [22], which we discussed in Section 7.2, could be used to present conflicts of ambiguous elements to the user. By showing all elements, which were matched by then (i.e., using the compare elements as first-class artifacts), the user could be able to manually decide how the conflict can be solved best. Another idea could be to first run the automatic decision wizard and show the possible solution for the conflict to the user. This way, the user can interfere with the automatic decision wizard to fix possible incorrect solutions before they are applied.

Bibliography

[1] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[2] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. "Clone Detection in Automotive Model-based Development". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. ACM, 2008, pp. 603–612.

[3] H. Frank and J. Eder. *Towards an Automatic Integration of Statecharts*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 430–445.

[4] H. Fuhrmann and R. von Hanxleden. "Taming Graphical Modeling". In: *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems*. MoDELS '10. Springer, 2010, pp. 196–210.

[5] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[6] S. Holthusen, P. Manhart, I. Schaefer, S. Schulze, C. Singer, and D. Wille. "Automatische Synthese von Familienmodellen durch Analyse von block-basierten Funktionsmodellen". In: *GI-Jahrestagung*. Vol. 220. LNI. in German. GI, 2013, pp. 2443–2457.

[7] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. "Family Model Mining for Function Block Diagrams in Automation Software". In: *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*. SPLC '14. ACM, 2014, pp. 36–43.

[8] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[9] R. Jockusch. "Verbesserung der Ähnlichkeitsbestimmung beim block-basierten Family-Mining". in German. Master's Thesis. Technische Universität Braunschweig, 09/2014.

[10] S. Lity. "Konzeption und Evaluation eines delta-orientierten modellbasierten Testverfahrens für Softwareproduktlinien". in German. Master's Thesis. Technische Universität Braunschweig, 12/2011.

[11] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. *Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study*. Technical Report. Technische Universität Braunschweig, 2012.

[12] C. Meyer. "Graphische Darstellung von Deltas in UML-Statechart Diagrammen". in German. Bachelor's Thesis. Technische Universität Braunschweig, 09/2014.

[13] T. C. Müller, M. Lochau, S. Detering, F. Saust, H. Garbers, L. Märtin, T. Form, and U. Goltz. *A Comprehensive Description of a Model-based, Continous Development Process for AUTOSAR Systems with Integrated Quality Assurance*. Technical Report. in German. Technische Universität Braunschweig, 06/2009.

[14] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. "Matching and Merging of Statecharts Specifications". In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. IEEE Computer Society, 2007, pp. 54–64.

[15] *OMG Unified Modeling Language (OMG UML), Superstructure*. Version 2.4.1. 2011.

[16] S. Oster, M. Zink, M. Lochau, and M. Grechanik. "Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations". In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. SPLC '11. ACM, 2011, 6:1–6:8.

[17] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. "Complete and Accurate Clone Detection in Graph-Based Models". In: *Proceedings of the 31th International Conference on Software Engineering*. ICSE '09. IEEE Computer Society, 2009, pp. 276–286.

[18] E. Rich. *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall, 2008.

[19] J. Rubin. *N-Way Model Merging. Project Report for CSC2125 Special Topics in Software Engineering: Modeling – Methods, Tools and Techniques*. Project Report. 2013.

[20] J. Rubin and M. Chechik. "Combining Related Products into Product Lines". In: *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 285–300.

[21] J. Rubin and M. Chechik. "N-Way Model Merging". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE '13. ACM, 2013, pp. 301–311.

[22] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. "A Relationship-driven Framework for Model Merging". In: *Proceedings of the International Workshop on Modeling in Software Engineering*. MISE '07. IEEE Computer Society, 2007.

[23] M. Stephan and J. R. Cordy. "A Survey of Methods and Applications of Model Comparison". In: *School of Computing, Queen's University, Kingston, Ontario, Canada, Technical Report* (2012).

[24] D. Wille, S. Holthusen, S. Schulze, and I. Schaefer. "Interface Variability in Family Model Mining". In: *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*. SPLC '13 Workshops. ACM, 2013, pp. 44–51.

[25] M. Zink. "Anwendung von MoSo-PoLiTe in einer automotive SPL". in German. Master's Thesis. Technische Universität Darmstadt, 06/2011.

A Appendix

This chapter contains the appendix of this thesis and provides further information on certain topics.

A.1. Used Key Values for ParameterizedElements

In Table A.1, we show a list of all `Strings`, that are used as keys for parameters in the `ParameterizedElement` class. The footnotes indicate, which parameters are used for which classes implementing the `ParameterizedElement` class. This list should be checked, before introducing new parameters to prevent conflicts and overriding parameters, that were previously set. The `KEY_VARIABILITY` parameter stores the variability, which is assigned to states, regions, and transitions. All parameters starting with `KEY_VARIABILITY_COUNT` store counters, which store the number of *mandatory*, *alternative*, and `optional` elements of a certain state chart element (i.e., states, regions, and transitions) in a model. The `KEY_GROUP` parameter and `KEY_GROUP_VARIABILITY` are used to store the group number of alternative groups and their variability (i.e., if they are optional or not).

Name	Used `String`
KEY_VARIABILITY[1, 2, 3]	"variability"
KEY_VARIABILITY_COUNT_STATE_MANDATORY[4]	"variability_count_state_mandatory"
KEY_VARIABILITY_COUNT_STATE_ALTERNATIVE[4]	"variability_count_state_alternative"
KEY_VARIABILITY_COUNT_STATE_OPTIONAL[4]	"variability_count_state_optional"
KEY_VARIABILITY_COUNT_REGION_MANDATORY[4]	"variability_count_region_mandatory"
KEY_VARIABILITY_COUNT_REGION_ALTERNATIVE[4]	"variability_count_region_alternative"
KEY_VARIABILITY_COUNT_REGION_OPTIONAL[4]	"variability_count_region_optional"
KEY_VARIABILITY_COUNT_TRANSITION_MANDATORY[4]	"variability_count_transition_mandatory"
KEY_VARIABILITY_COUNT_TRANSITION_ALTERNATIVE[4]	"variability_count_transition_alternative"
KEY_VARIABILITY_COUNT_TRANSITION_OPTIONAL[4]	"variability_count_transition_optional"
KEY_GROUP[1, 2, 3]	"Group"
KEY_GROUP_VARIABILITY[1, 2, 3]	"GroupVariability"

[1] Assigned to `State` objects
[2] Assigned to `Region` objects
[3] Assigned to `Transition` objects
[4] Assigned to `StateChart` objects

Table A.1.: A list of all `Strings` used for parameters in `ParameterizedElements`

A.2. Used Key Values for Metrics

In Table A.2, we present a summary of all key values, which are used in the abstract `Metric` class. Besides, we show a number, which shows the create method, that needs to implement this weight. These numbers, the corresponding creation methods, and a short explanation for them can be found in Table A.3.

Key value	Explanation	Method #
STATE_STATIC_NAME	Weight for the state name	1
STATE_STATIC_START_STATE	Weight for the "initial state" property	1
STATE_STATIC_END_STATE	Weight for the "end state" property	1
STATE_STATIC_PARALLEL_STATE	Weight for the "parallel state" property	1
STATE_STATIC_HIERARCHY_DISTANCE	Weight for the calculated hierarchy distance	1
STATE_STATIC_NEIGHBOR	Weight for the similarity of the neighbors	1
STATE_STATIC_STEREOTYPE	Weight for the stereotype of the state	1
STATE_DYNAMIC_HISTORY_STATE	Weight for the "history state" property	2
STATE_DYNAMIC_DEPENDENT_ON	Weight for the *dependent on* actions	2
STATE_DYNAMIC_TRIGGERED	Weight for the *triggered* actions	2
STATE_DYNAMIC_TRIGGERING_CHANGE	Weight for the *triggering change* actions	2
STATE_NEIGHBORHOOD_NEIGHBOR_SIMILARITY	Weight for the similarity of all neighbors in the neighborhood	3
STATE_NEIGHBORHOOD_INTERFACE_SIMILARITY	Weight for the similarity of the compared interfaces	3
STATE_NEIGHBOR_NAME	Weight for the names of compared neighbors	4
STATE_NEIGHBOR_ACTIONS	Weight for the actions of compared neighbors	4
TRANSITION_STATIC_NAME	Weight for the transition name	5
TRANSITION_STATIC_STEREOTYPE	Weight for the stereotype of the transition	5
TRANSITION_DYNAMIC_TRANSITION_LABEL	Weight for the similarity of transition labels	6
TRANSITION_DYNAMIC_TYPE	Weight for the "transition type" property	6
TRANSITION_DYNAMIC_PRIORITY	Weight for the transition priority	6
TRANSITION_LABEL_EVENT	Weight for the events	7
TRANSITION_LABEL_CONDITION	Weight for the conditions	7
TRANSITION_LABEL_CONDITION_ACTION	Weight for the condition actions	7
TRANSITION_LABEL_TRANSITION_ACTION	Weight for the transition actions	7
REGION_SUBSTATE	Weight for all region sub-states	8
REGION_SUBTRANSITION	Weight for all region sub-transitions	8
STATE_STATIC	Weight for all *static* state attributes	9
STATE_DYNAMIC	Weight for all *dynamic* state attributes	9
TRANSITION_STATIC	Weight for all *static* transition attributes	10
TRANSITION_DYNAMIC	Weight for all *dynamic* transition attributes	10

Table A.2.: A list of all key values for metrics

Method #	Name	Explanation
1	`createStateStaticElementsMap()`	Provides the weights for the *static* state attributes
2	`createStateDynamicElementsMap()`	Provides the weights for the *dynamic* state attributes
3	`createNeighborhoodInterfaceElementsMap()`	Provides the weights to compare the interfaces of states
4	`createStateNeighborElementsMap()`	Provides the weights for the comparison of state neighbors
5	`createTransitionStaticElementsMap()`	Provides the weights for the *static* transition attributes
6	`createTransitionDynamicElementsMap()`	Provides the weights for the *dynamic* transition attributes
7	`createTransitionLabelElementsMap()`	Provides the weights to compare labels with each other
8	`createRegionSubElementsWeightsMap()`	Provides the weights for the states and transitions in regions
9	`createStateWeightsMap()`	Provides the weights for *static* and *dynamic* state parts
10	`createTransitionWeightsMap()`	Provides the weights for *static* and *dynamic* transition parts

Table A.3.: A list with all metric creation methods and their number in Table A.2

A.3. Graphics of the BCS SPL case study

In Subsection 6.1.2, we already presented the graphics for the fine granular and large variation points of the *BCS SPL case study*. In this section, we present the remaining graphics of the 150% state chart for the *BCS SPL case study*, which do not contain any variation points.

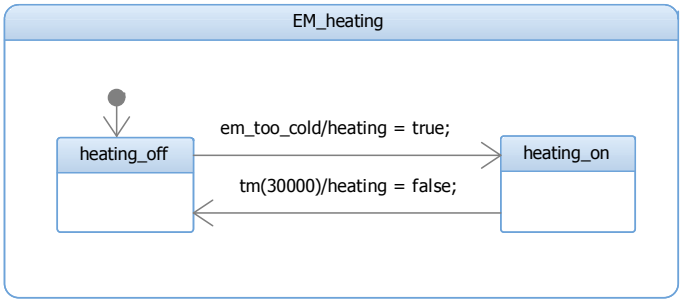


Figure A.1.: Contents of the `EM_heating` state in the *BCS SPL case study*

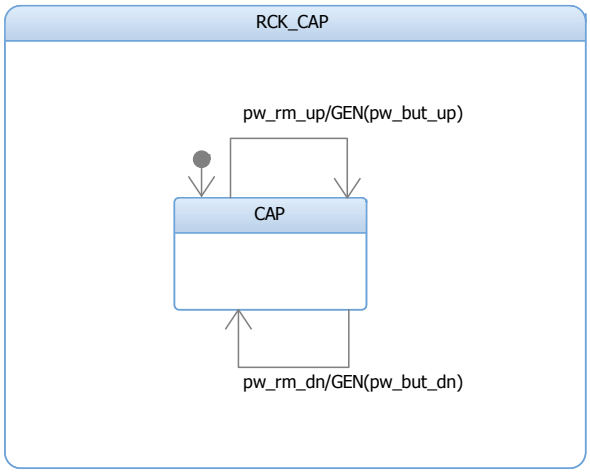


Figure A.2.: Contents of the `RCK_CAP` state in the *BCS SPL case study*

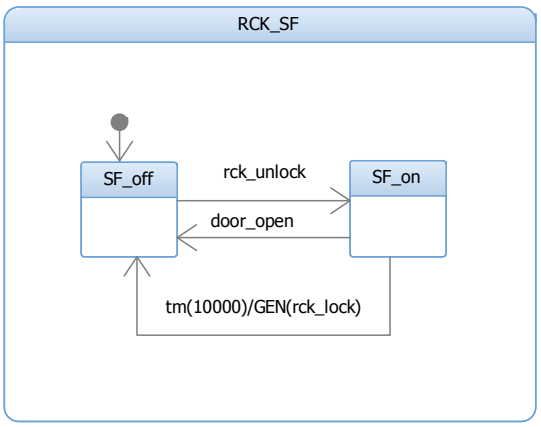


Figure A.3.: Contents of the `RCK_SF` state in the *BCS SPL case study*

Figure A.4.: Contents of the AutoPW state in the *BCS SPL case study*

Figure A.5.: Contents of the ManPW state in the *BCS SPL case study*

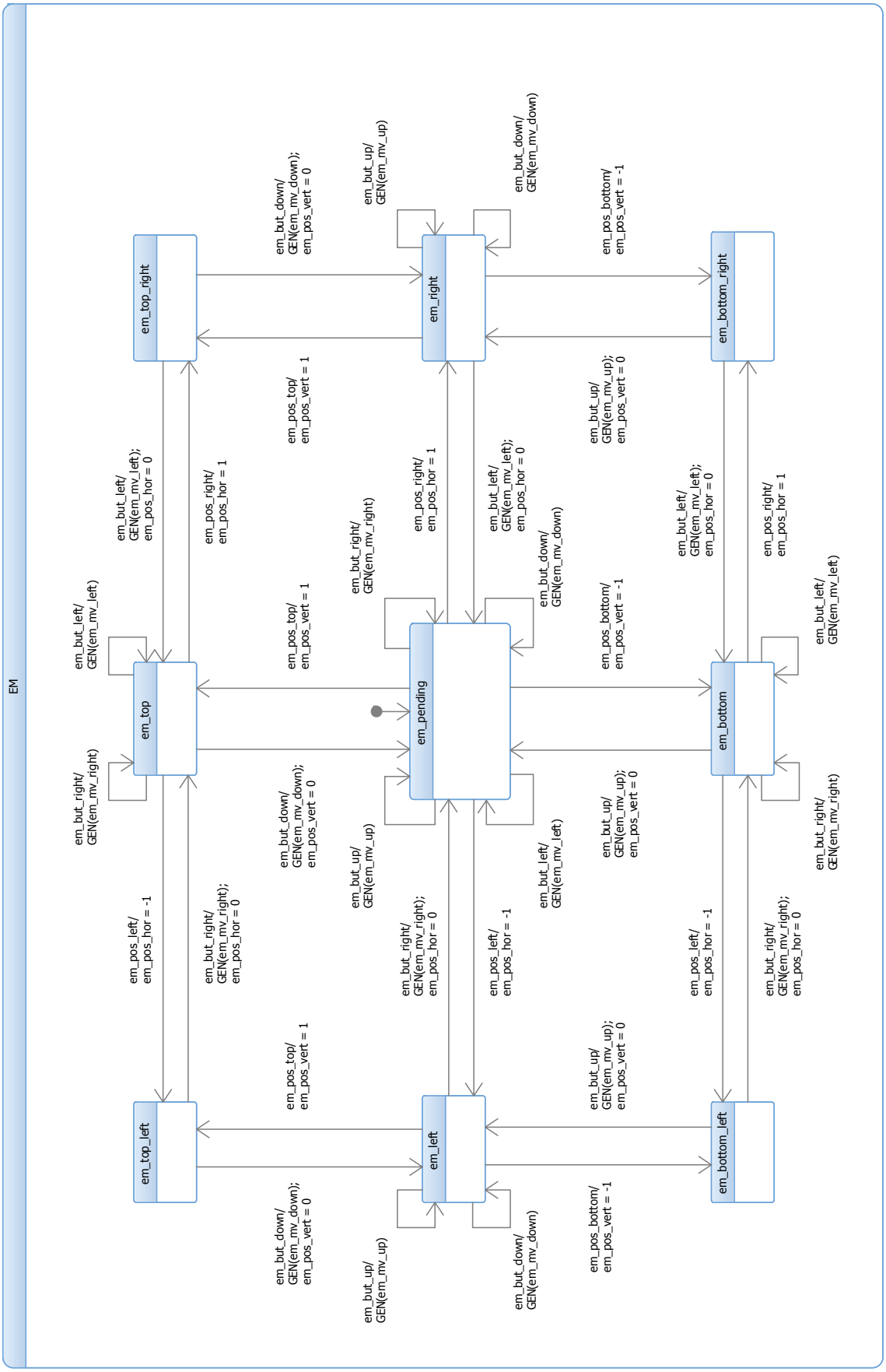


Figure A.6.: Contents of the EM state in the *BCS SPL case study*

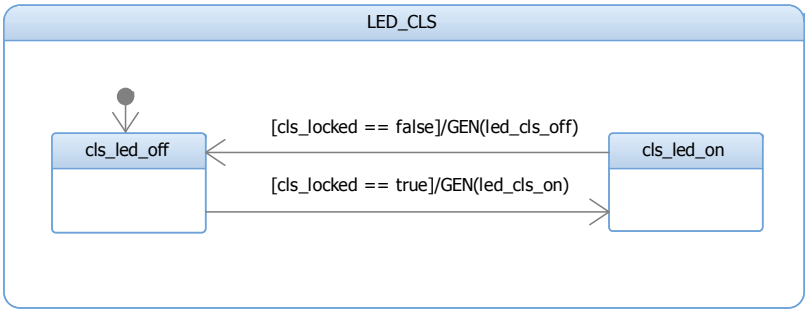


Figure A.7.: Contents of the LED_CLS state in the *BCS SPL case study*

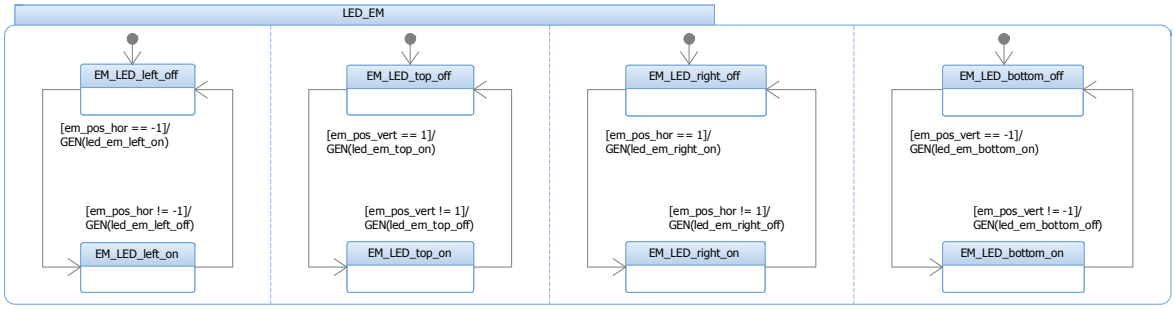


Figure A.8.: Contents of the LED_EM state in the *BCS SPL case study*

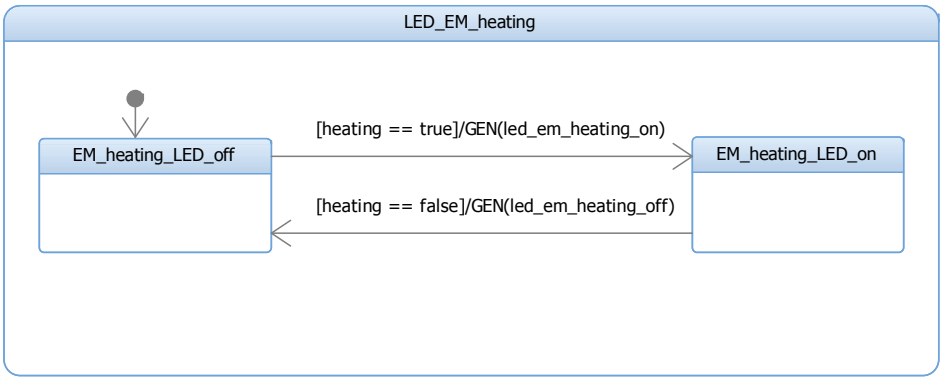


Figure A.9.: Contents of the LED_EM_heating state in the *BCS SPL case study*

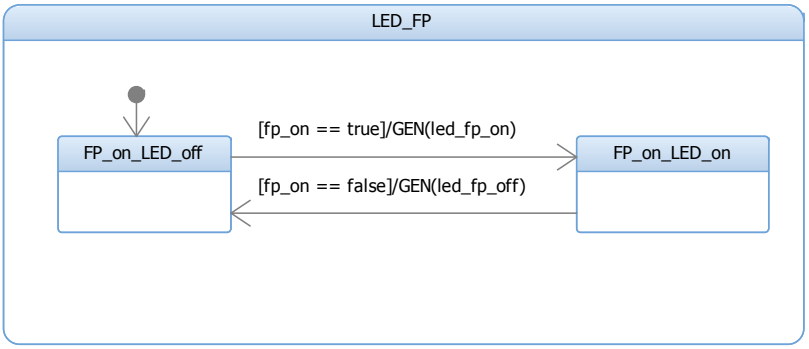


Figure A.10.: Contents of the LED_FP state in the *BCS SPL case study*

A.4. Overview over the Product Configurations of the BCS SPL case study

In Table A.4, we present the overview over the product configurations of the *BCS SPL case study*, taken from [11]. This table helps with the analysis of the different state chart variants for these products.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
Security		✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	
Status LED		✓		✓		✓			✓	✓	✓	✓		✓			✓	
LED Central Locking System				✓		✓	✓			✓		✓		✓			✓	
LED Power Window				✓		✓	✓		✓	✓		✓						
LED Heatable				✓		✓	✓		✓	✓				✓			✓	
LED Exterior Mirror				✓		✓	✓		✓	✓	✓						✓	
LED Alarm System				✓		✓	✓			✓		✓					✓	
LED Finger Protection		✓				✓	✓		✓	✓	✓	✓		✓			✓	
Manual Power Window	✓			✓		✓	✓			✓	✓	✓		✓			✓	✓
Automatic Power Window		✓	✓		✓			✓	✓	✓			✓		✓	✓		
Heatable		✓		✓		✓	✓		✓	✓			✓	✓	✓	✓	✓	✓
Central Locking System		✓	✓	✓	✓		✓		✓	✓		✓	✓	✓	✓	✓	✓	
Remote Control Key		✓	✓	✓	✓				✓	✓		✓	✓	✓	✓	✓	✓	
Alarm System		✓	✓	✓	✓	✓	✓		✓	✓		✓			✓	✓	✓	
Automatic Locking			✓		✓		✓					✓	✓	✓	✓	✓	✓	
Control Alarm System		✓	✓	✓					✓			✓			✓	✓	✓	
Safety Function			✓		✓				✓	✓		✓	✓	✓	✓		✓	
Control Automatic PW			✓		✓				✓	✓			✓		✓	✓		
Interior Monitoring		✓			✓	✓	✓		✓			✓			✓	✓	✓	

Table A.4.: Overview over the product configurations of the *BCS SPL case study*, taken from [11]

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)
Institute of Software Engineering and Automotive Informatics

Mühlenpfordtstr. 23
D-38106 Braunschweig